Coordinating exploitation of data and control parallelism

Sonia Campa

20th January 2006
Thesis Supervisor:
Marco Danelutto

External Reviewers:
Murray Cole and Jocelyn Serot

Internal Committee:
Gianluigi Ferrari and Paolo Mancarella
Abstract

The aim of this work is to introduce a new programming model in which parallel applications are built by keeping data and control parallelism concerns orthogonal through a set of abstract mechanisms. We provide mechanisms for abstracting on the data accesses as well as mechanisms for abstracting from communication patterns, taking inspiration from the skeletal programming approach. The point of contact between the two abstract mechanisms that allows us to coordinate such orthogonal aspects is represented by the concept of iterator. We have been inspired by the design pattern introduced by Gamma et al. and we have extended its semantics by means of a set of behavioral properties.

After a first formalization of the model, we concentrate our attention on the evaluation of a parallel application which is described by a sequence of transformations (i.e. inference rules) on the application graph. The contribution given by this piece of work is a more clear specification of the semantic framework and the introduction of a computational costs system provided by such semantics model. The former formalizes the mechanisms for describing both control and data parallel concerns, allowing to describe the evaluation of an application graph. The semantic framework exposes a set of semantics expressions that define our abstract mechanisms and a set of 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 programming model and semantics capabilities of the costs system have been proved through a Java implementation framework on which some sample examples and a Genetic Programming application have been implemented. Such tests have demonstrated the expressive power of our approach and the potentiality of the associated semantic framework and computational costs system.
Acknowledgments

This work has been developed in a very particular period of my life where certainties continuously alternated with uncertainties about myself, my nature and my wishes. It has been a hard work that probably I had never completed without the encouragement coming from some very special people. First of all, this work is dedicated to Salvo: his beautiful personality gave more sense to all this path, lighted heavy days and expanded pleasant feelings. He contributed in make me stronger and motivated in every day life, he shared and still shares with me crazy projects and hopes for the future. A special mention goes to my family, always in mind, in my heart and in my soul for their trust, their silent support and the possibilities they provide me until now. Thanks to my brother, for every word, for every thought. Hernan, Francesca, Miriam demonstrated to be great friends, besides colleagues and people with which I had a very good time and discussions. I hope to share with them more and more time in the future.

A special thanks goes also to one of the best friend I’ve never met, Flavio, for his wonderful ability in transforming every dark feeling in a funny joke through his irony and free spirit that have inspired me from the first day I met him.

Obviously I couldn’t finish this work and this piece of life without the support of my supervisor Marco Danelutto, to which goes all my gratefulness for the opportunities, for the time and, last but not least, for his precious friendship. Thanks to Marco and Paolo for their suggestion about my work and thanks for being so kind and friendly with me.

And thanks to Morgan and Kleopatra, for their lovely purrs and the pure joy they gave me in every single moment I spent with them and their irresistible, furry muzzles. Even if it could sound strange, they helped me in affording the worst crisis by simply living close to me.
To Salvo and to my family
1 Introduction 9
  1.1 Motivations .......................................................... 11
  1.2 At first glance ......................................................... 13
  1.3 Thesis plan ............................................................. 18
2 Background 19
  2.1 Coordination models and languages ............................... 19
  2.2 Classification ........................................................ 21
    2.2.1 The data-driven model ................................... 22
    2.2.2 The control-driven model ................................ 23
  2.3 Coordination models for orthogonal data and control parallelism handling ........................................ 25
    2.3.1 Data-driven approaches ................................... 25
    2.3.2 Control-driven approaches ................................ 27
    2.3.3 Hybrid approaches ........................................... 28
  2.4 Discussion ............................................................. 31
3 The programming model: an overview 35
  3.1 Abstraction mechanisms ........................................... 35
  3.2 Separation of concerns ............................................. 37
  3.3 Writing an application ............................................. 38
  3.4 The semantics model ................................................ 39
4 Accessing data 41
  4.1 Abstract Data Type .................................................. 42
  4.2 Views ................................................................. 43
    4.2.1 ArrayView ..................................................... 43
    4.2.2 MatrixView ................................................... 43
    4.2.3 ListView ....................................................... 44
    4.2.4 GraphView ..................................................... 44
    4.2.5 TreeView ....................................................... 45
  4.3 Characterizing the access through patterns of access ........ 46
    4.3.1 Array Pattern .................................................. 46
    4.3.2 Pattern for accessing matrices - ByPointPattern ....... 47
    4.3.3 Pattern for accessing matrices - ByRowPattern ........ 48
    4.3.4 Pattern for accessing matrices - ByColPattern ......... 49
  4.4 Iterators ............................................................. 50
  4.5 Related work ........................................................ 51
5 The semantic framework: control parallelism 53
  5.1 Control primitives .................................................. 53
  5.2 A type system of control primitives ............................. 54
    5.2.1 Primitive Fun ................................................ 55
10 Experimental results

10.1 The experiments
10.1.1 Apply(seqdo) performance
10.1.2 seqdo(Apply) performance
10.1.3 Apply(Pipe) performance
10.1.4 Pipe(Apply) performance
10.2 Rewriting rules exploitation
10.3 Comparisons between estimated and actual computational costs
10.4 A case study: Genetic Programming
10.4.1 A benchmarks for GP: the Even $k$-parity problem
10.4.2 Parallel and distributed Genetic Programming
10.5 Even $k$-parity problem implementation
10.5.1 Impl2 version
10.5.2 Impl3 version
10.5.3 Computational costs comparison
10.6 Conclusions

11 Conclusions and future work

Bibliography
List of Tables

4.1 A grammar defining the $A$ type ........................................ 41
4.2 Operators on ADT's ..................................................... 42
4.3 Operators on ArrayView ................................................. 43
4.4 Operators on MatrixView ............................................... 44
4.5 Operators on ListView .................................................. 44
4.6 Operators on GraphView ................................................ 45
4.7 Operators on TreeView .................................................. 46
4.8 Pattern of access on the set of views ............................... 47
4.9 Array Pattern ............................................................. 47
4.10 ByPointPattern operators ............................................... 48
4.11 ByRowPattern operators ............................................... 49
4.12 ByColPattern operators ............................................... 49
4.13 Operators for the $I$ type ............................................. 51
6.1 Evaluation rules on the iterator data type ......................... 62
6.2 Transformation rules for reducing seqdo .......................... 63
6.3 Transformation rules for reducing pardo .......................... 64
6.4 Transformation rule for reducing primitive Fun. .................. 64
6.5 Transformation rule of primitive Seq ............................... 65
6.6 Transformation rule of primitive Apply ............................ 66
6.7 Transformation rules for evaluating primitive Pipe ............. 68
7.1 Cost system for pardo ................................................... 73
7.2 Cost system for reducing seqdo ...................................... 74
7.3 Cost system for iterator type ......................................... 74
7.4 Cost system for primitive Fun ....................................... 75
7.5 Cost system for the Apply primitive ............................... 75
7.6 Transformation steps for Apply given a parallel iterator ....... 76
7.7 Transformation steps for Apply given a sequential iterator ... 76
7.8 Computational costs for primitive Seq ............................ 77
7.9 Cost system for the primitive Pipe .................................. 78
7.10 Summary of the computational costs per primitive ............. 81
10.1 Characteristics of cluster pianosa.di.unipi.it ...................... 119
10.2 Truth table of the optimal individual for the even 2-parity problem .... 132
10.3 Truth table of the individual $e = \text{or}(\text{and}(x_1,x_2),\text{and}(x_1,x_1))$ ....... 133
Chapter 1

Introduction

With the evolution of distributed and parallel systems, new programming languages were developed to make use of the available parallelism and of the (often) massive number of processors composing a system. These languages were able to exploit parallelism, perform communication and be fault tolerant. They differed in the granularity or unit of parallelism they exploited (e.g. sequential processes, objects, parallel statements etc.) and in the communication mechanism they employed (e.g. message passing models such as rendez-vous or remote procedure calls, data sharing models such as distributed data structures or shared variables, etc.).

The increased availability of massively parallel and open distributed systems lead to the design and implementation of complex and large applications such as vehicle navigation, air traffic control, intelligent information retrieval and multimedia based-environments, to name some of them.

Gradually, it became evident that no unique programming language or machine architecture was able to deal in a satisfactory way with all the facets of developing complex and multi-functional applications.

Issues such as reusability, compositionality and extensibility became of paramount importance.

The new scenario depicted by the availability of an increasing number of different architectures and, on the other hand, the need for supporting a wider and wider range of programming languages leads to a new requirement for parallel applications: portability. As a consequence, providing mechanisms (at compile or run time) that give the possibility to execute the same code on different architectures and/or operating systems with acceptable performance becomes a hot research topic.

Real systems are composed of components that interact with each another and each component is an open system in isolation. Typically, the behavior of each of these components is ill-defined, except within confines of a set of constraints on the surrounding interactions with its environment. When a number of such open systems come together as components to build a larger system, the topology of their interactions forms a context that constrains their natural interactions and yields well-defined behavior. In other words, each component interacts with the other components by following rules (or communication paradigms) that are given and known inside the surrounding context. Moreover, since each components adhere to common interaction rules, the well-behave of the whole system is ensured.

On the other hand, one can abstract away from all such interactions, regarding them as internal details of individual components, focusing on a formal study of the constraints, context and conditions on the interactions among the components in a system (as well as between the system and its environment) that ensure and preserve well-behaves.

In order to deal with all these requirements of programming-in-the-large applications, the notions of multilingual environments and heterogeneity come into play. *Multilingual* or *multi-paradigm* programming is able to support a number of paradigms and provides inter-operation of these paradigms and at the same time, isolates unwanted interactions among them. A multilingual or multi-paradigm programming environment aims at accommodating the diverse execution models and mechanisms of the various paradigms, manages the resource required
for implementing these paradigms. Moreover, it offers intuitive ways for combining code written in a mixture of paradigms providing to the user orthogonal programming interfaces, at the same time.

There are basically two ways to produce multilingual or multi-paradigm languages:

- designing a new “super-language” that offers the facilities of all paradigms intended to be used.
  This approach has the advantage of usually providing a more coherent combination of different paradigms but the programmer has to learn just another programming language that cannot possibly support all the functionalities of the languages it aims to replace;

- providing an interface between existing languages.
  This second approach can be realised with different degrees of integration ranging from using the operating system’s communication primitives for inter-component collaboration to providing concrete integration of the various languages involved.

Multilinguality is closely related to heterogeneity since heterogeneous systems demand that a programming language must be able to express many useful models of computation. In fact, the concept of heterogeneity can be afforded at different levels of abstraction:

- hardware architectures can be heterogeneous, meaning that the application (or parts of it) will run on different processing elements and/or that different resource will be involved by the execution. As an example, applications running on Grid architectures are exposed to hundred or thousands of different kind of processors, each “talking” its own language.

- heterogeneity can also arise at software level. In particular, operating systems are today widely heterogeneous, thus requiring plain compatibility of the application with the underlying OS standards.

- heterogeneity can be exploited by the application itself. In fact, each parts of it may require different programming models, ranging from a distributed data space shared with other applications in the net to message passing mechanisms to interact with the outer world.

However, it is usually impossible to find a single language able to deal satisfactorily with an extensive variety of such models; a mixture of language models may have to be employed.

In the field of massively parallel systems a further question is how to ensure proper communication among the hundreds and thousands of different pieces of code that comprise the active entities in a single application [CG89a]. The difficulty in answering this question have limited the practical use of MIMD\(^1\) while, on the other hand, has lead to a growing research effort in the direction of SPMD models, a restriction of the MIMD model.

A SPMD\(^2\) model simplifies the problem of concurrency. Logically, it allows several processors, all executing the same program but on different data, proceed at their own pace up to a common barrier (the only coordination construct), where they synchronize. The barrier can be released provided that the asynchronous processors synchronize when they access shared variables. Nevertheless there are applications that don’t fit in the uniformity offered by the SPMD model and require more flexible coordination. Examples include computations involving large dynamic trees, symbolic computation on parallel machines, and dynamic pipelines. Taking full advantage of the potential offered by massively parallel systems in these and other applications requires massive, non-replicated patterns of concurrency.

Besides multilinguality and heterogeneity, another aspect influencing the design of a parallel programming model is the way parallelism is expressed. In fact, by means of a distinction close to the one given between MIMD and SPMD models, we can classify most of the parallel programming systems as based either on control parallelism or on data parallelism [BH98, ST98].

---

\(^1\)Multiple Instruction Multiple Data
\(^2\)Single Program Multiple Data
1.1. MOTIVATIONS

Control parallelism  Control parallelism (also known as task or process parallelism) allows programmers to define different type of processes. These processes communicate and synchronize with each other through message passing or other mechanism. In between synchronization points, each process executes independently from all the other processes. With a task parallel programming system the programmer must deal explicitly with process creation, communication and synchronization.

Data parallelism  Data parallelism is a quite different model and is based on applying the same operation in parallel on different elements of a data set. Unlike control parallelism, all processors conceptually execute the same program, on different data elements.

The advantage of data parallelism is that it uses a simpler model and the task of the programmer can be reduced substantially by providing the right language and compiler support: in HPF, for example, the programmer is mainly responsible for specifying the distribution of data structures. The compiler takes care of generating the necessary code for communication and synchronization.

1.1 Motivations

Unfortunately, the simple model of data parallelism also makes it less suitable for applications that do not fit the model. In particular, applications that use irregular data structures often do not match the model and impose difficult problems on both the language designer and compiler writer. It is widely accepted that many important parallel applications cannot be efficiently implemented following a pure data parallel paradigm: pipelines of data parallel task, that is a common computation structure in image processing, signal processing or computer vision; irregular application as the ones working on dynamic data structures (trees, graphs, etc.); multidisciplinary optimization problems like aircraft design, etc. For these applications, rather than having a single data parallel program, it is more appropriate to subdivide the whole computation into several data parallel pieces, that run concurrently and co-operate, thus exploiting task parallelism, too.

These observations have motivated many proposals for the integration of task and data parallelism (see Section 2.3.3).

[BH98] identifies at least three advantages for integrating task and data parallelism within the same language.

1. generality: given the large number of parallel programming models and languages available, having a single language that is capable of encoding a wide-variety of parallel applications is certainly attractive;

2. increasing scalability by exploiting both forms of parallelism within a single application. For many applications, data sizes are fixed and cannot easily be increased, thereby limiting the amount of data parallelism that can be exploited.

3. coordination of multidisciplinary applications: many modern scientific applications are created out of a collection of subprograms from a variety of different disciplines that are integrated into a single, multidisciplinary application. To efficiently coordinate the execution of these independent data parallel models (which can be viewed as very large-grained tasks) a language that supports both task and data parallelism is required.

Although several approaches have been proposed, integrating the two forms of parallelism cleanly and within a coherent programming model is difficult because of the way parallelism and communication are expressed in the two models and the implementation: data parallelism focuses on data, control parallelism focuses on interactions among processes.

In fact, generally speaking, data parallel languages produce a single program that is executed on all machines, resulting in a SPMD model. In contrast, task parallel programs typically consist of many different types of processes that execute largely independently from each other. Such programs are written in an MIMD style. An integrated model should support both styles (MIMD
and SPMD), but clearly they are quite different, such that most languages are biased either to an SPMD or MIMD style.

A second key difference between task and data parallel languages concerns the organization of the address space. In modern data parallel languages parallelism occurs in a single, global, shared address space. In a distributed implementation, the compiler takes care of generating code for data transfer, transparently to the user. Many task parallel languages, on the other hand, provide a separate address space for each process, and require the programmer to insert explicit send and receive statements to transfer data between these disjoint address spaces. Again, this difference makes the smooth integration of task and data parallelism difficult.

Not all task parallel languages use multiple address space. Distributed Shared Memory (DSM) systems provide a logically shared address space on top of a distributed-memory architecture in which it may be easier to integrate with data parallelism than message passing systems, just because the first focuses on data access.

A fourth difference between task and data parallel languages is the way they are implemented. Data parallel languages like HPF rely on extensive compiler analysis, for example to compute efficient communication schedules or to vectorize multiple data transfer using a single message. Consequently these languages are designed to allow such analysis to be done statically. In particular, the data distribution is usually known at compile-time.

Task parallel languages use traditional compiler technology designed for sequential languages. Unlike data parallel languages, however, they often use complicate runtime systems. Even traditional message passing languages usually have extensive runtime systems, that buffer, order and filter incoming messages.

Since task parallel languages do not make high demands on the compiler, they often have fewer restrictions than data parallel ones. Typically, they allow the dynamic creation and allocation of processes onto processors. If such task parallel constructs are added to a data parallel language, it will make compile-time analysis much more difficult, if not impossible. To prevent this, some languages impose restrictions on the task parallel constructs and require processor allocation to be specified statically. An interesting alternative would be to use run-time compilation, but this technique is exploited in few system yet.

Thus, from the point of view of implementation, two principal approaches have been investigated for the integration of task and data parallelism. Compiler-based approaches that seek to identify task-parallel structures automatically, within data-parallel specifications; language-based approaches that, instead, provide new language constructs for specifying control parallelism explicitly. Both approaches have raised promises in certain application areas, but each also has disadvantages. Compiler-based approaches complicate compiler development and performance tuning, and language-based approaches also introduce the need to standardize new language features.

Several distinct parallel programming models recognized that parallelism exploitation may be better organized using compositions of instances of items belonging to a basic set of parallelism exploitation patterns, rather than explicitly using the mechanisms provided by the programming environment for parallel activities setup (processes, threads), for communications (point-to-point, collective), for mapping and scheduling of parallel activities. In particular, the algorithmical skeleton research track originated by Cole’s works \cite{cole89, ske, dop91, dt02, ds01, dgyt95, SGD99}, the research track related to design patterns \cite{bma02, fos95, ghjv95, mms01} and the track related to coordination languages \cite{ach98, ah93, ch96, pa97}, all individuated several standard methods (skeletons, design patterns, coordination patterns) that can be used to build a significant part of the interesting, scalable parallel applications currently required.

Some of the proposed programming models in this framework(s) explicitly assume that the parallelism exploitation methods can be arbitrarily composed \cite{bdpv99, dop91}. Other models assume they can’t be composed at all or they can be composed in rather limited forms \cite{bk96, hb398, mb96, van02}. Furthermore, some of the models provide users/programmers with both data and control (or task) parallel exploitation mechanisms (design patterns, coordination patterns, skeletons) \cite{bdpv99, dpp97, drst01, kuc02, mss99} whereas other models just provide either
control or data parallel exploitation mechanisms [ACG86, For97, GC92]. Some works concentrate on the need of integrating data and control parallelism into a well defined model [BH98, DRST01, FKKC97, KC02]. The results of such attempts often led to the design of two-tier models which the lower level exposes task parallelism by structuring control flow trees. Each leaf of a control tree represents a sequential function or a data parallel operation.

Two-tier models have lead the development of several libraries and frameworks [Pel02, CLS+00] but none of them has been able to define a clear, powerful, well defined separation of concerns. This is a key point, indeed. Different classes of parallel applications can be exploited depending on the kind of parallelism exploitation patterns provided to the programmer: data parallel only, control parallel only, both. Furthermore, usually data and control parallelism exploitation mechanisms are provided in quite different flavors. Control parallel mechanisms, such as pipelines and task farms, are usually provided by means of some kind of parallel control statement parametric in the pipeline stages or task farm worker computations. Data parallel mechanisms, such as independent or non independent forall or apply-to-all, reduce, prefix, etc. are defined in terms of some kind of parallel (array) data structure. As a consequence, the semantics of joint data and control parallel applications is difficult to express and to use to derive program transformation or program verification techniques, which are usually very useful to support performance tuning as well as application optimizations.

In this thesis we will present a structured parallel programming model that clearly separates data parallel and control/task parallel mechanisms in such a way they can be orthogonally composed to build more complex parallelism exploitation patterns while preserving the clear semantic properties deriving from the separation of concerns. Such orthogonality supported with a well-founded semantics and a set of rewriting rules could provide a methodological basis for writing efficient and portable parallel programs implemented by high performance implementation tools.

1.2 At first glance

One of the most relevant trouble in writing parallel programs from the user’s point of view is given by the need of expressing both data and control parallel concerns at the same time. Expressing data parallelism leads to the definition of how data that could be processed 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 item has to be processed. This concerns are intrinsically orthogonal. Nevertheless, writing a parallel application still means keeping them together in mind, in a mixture of data accessing and control code that is often difficult to coordinate.

The conceptual 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 composable and nestable 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 graph is formally described by of 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 four mechanisms: abstract data types, views, access patterns 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. For instance, the set of objects

\[ s = \{x_1, \ldots, x_n\} \]

represents a raw input dataset of \( n \) values.
Moreover, the model provides different kinds 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 of its actual implementation. For instance, an array view could be defined on top of a set of items evaluating it as a collection of elements accessible by referring their position in the collection. So, taking the abstract data type \( s \) defined above, an array view \( v \) defined on top of \( s \), could allow to access each single element by specifying its position.

Further, the programmer could apply different kinds of views on the same abstract data type in order to take advantage of different logics and operators to manage its data. Since these are logical transformations, the framework will evaluate and optimize the actual implementation of data, views and operations on them. As an example, an array view could be applied on the abstract data type \( s \) defined above, as well as a matrix. In the latter case, each row of the matrix is represented by a contiguous block of elements of \( s \).

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 singletons, 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.

The behavior of an iterator instance depends on a view and on the particular access pattern defined on that view. An access pattern describes the minimal unit of access for a given view. As an example, on an array view, the minimal unit of access can be a single element of the array or a block of \( k \) elements. In the first case, we will declare an access pattern \( P \) of grain 1, otherwise we will declare \( P \) of grain \( k \). Each iterator defined on top of the array view and coupled with the access pattern \( P \) will access the view whose granularity depends on the minimal unit represented by \( P \).

Let us take into account the array view \( v \) given above and suppose that the its length \( n \) is a multiple of 2. Let \( it \) be a sequential iterator associated to \( v \) and \( a \) be the access pattern of grain 2 defined on top of \( v \). Then, the array view \( v = [s_1, \ldots, s_n] \), where each \( s_i \) is an element of the input data set, will be accessed by blocks of 2 elements \([s_i, s_{i+1}]\) for all \( i \in [1, n] \) (because of the access pattern type) and such blocks will be provided to the programmer sequentially (because of the iterator type).

The control behavior of the whole application is described by instantiating a set of control primitives that couple each selected control pattern with one or more iterators. The internal parallel behavior of each primitive 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 a matrix view. The application involves only one pattern of control (it is called Apply and applies a function to all the items of a set) and the user can decide if instantiating an iterator that returns the items in a whole or one-by-one on its matrix view. In case of the first choice, combining the iterator with the Apply control pattern will exploit a plain data parallel behavior, in the second case the combination acts as a plain task parallel behavior.

Using such primitives 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 proved by the framework through a formal semantics that describes each entity involved in the parallel application.

**The semantic framework** The formalization given in this thesis to the abstract mechanisms described above, starts from a global view of the programming model \( M \). In fact, it can be completely defined as a tuple

\[
M = \langle A, V, P, I, C \rangle
\]

where \( A \) stands for a set of abstract data types representing the raw input dataset, \( V \) is the set of view types, \( P \) is the set of access patterns, \( I \) represents the iterator type and \( C \) is the set of collective primitives.
Each member of the tuple is described by a semantic that formally defines it. Abstract data types are described as a collection of ground or abstract data type values, on which typical operations on set can be applied. As an example the set

\[ s = \{ s_i | i \in [0, n - 1] \land s_i = 2 \times i + 1 \} \]

is the set representing the first \( n \) positive odd numbers.

Views are defined as structured set of values built on top of a given abstract data type. As an example, an array view defined on top of \( s \) is described as an homomorphism between elements of \( s \) and elements of \( v \), thus

\[
\text{ArrayView}(s) = \{ v_i | \forall i \in [1, \text{size}(s)].v_i = s_i \}
\]

Defining an access pattern on top of a view means refining the description of the view access by defining the structure of a single unit. An array view can be accessed by single positions or by blocks of contiguous elements, thus changing the access grain. As a consequence, an access pattern on this kind of view allow to define the granularity by which each element of the view should be access. \text{ArrayPattern} is the name of the access pattern associated to the array view and its definition takes the view and a grain value as input, thus

\[
\text{ArrayPattern}(v, g) = \{ a_i | \forall i \in [0, \frac{\text{size}(v)}{g} - 1].a_i = v_i \times g; \ldots; v_i \times g + (g - 1) > \}
\]

provides the formal definition of this pattern.

Each view is a factory for a set of iterators on its structure. Generally speaking, an iterator is a couple \(( ap, p )\) where \( ap \) represents the associated access pattern on the given view and \( p \) is a pointer to the next accessible item of its set of elements. Formally the iterator type is defined as

\[
I : \text{AccessPattern} \times \text{Pointer}
\]

where \( \text{Pointer} \) represents a generic pointer type on the set of elements provided by the access pattern.

An iterator object of type \( I \) is provided with a set of operators for getting and settings elements coherently with both the access rules provided by the encapsulated access pattern and its own type (sequential or parallel). One of the most important operator is called \text{current()} and returns the current item pointed by \( p \). 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.

Using \( A, V, P \) and \( I \) objects and its operators allows to describe how to access the input data. Thus, in a word, they allow to describe exclusively data parallel concerns.

In an orthogonal, independent manner, it is possible to describe control patterns. In fact, the framework provides a set of primitives that can be combined and nested in order to built the whole application graph.

At the moment, the set of primitives is defined by

\[
C = \{ \text{Fun, Seq, Pipe, Apply} \}
\]

where \( \text{Fun} \) represents a pattern embedding a user-defined sequential function; \( \text{Seq} \) presents the sequential composition of two or more inner primitives; \( \text{Pipe} \) represents the pipelining of a set of embedded primitives; \( \text{Apply} \) represents the application of an inner primitive to a set of input values.

The evaluation (i.e. the execution) of an application graph is done by coupling each primitive composing the graph with a given iterator. In fact, as the combination of primitives represents the control concerns of the application, the parallel access behavior by which the input dataset will be accessed is encapsulated by the coupled iterator instance: this means that the programmer can concentrate on the structural aspects of his computational graph, regardless how to deal with data parallel concerns.
Each primitive evaluation is represented by a set of inference rules describing how the primitive evolves in time. Such evolution will depend both from the type of primitive (i.e., it could encapsulate an inner primitive that requires to be evaluated, first) and from the type of the coupled iterator. Generally speaking, an inference rule $T_0$ has the following structure

$$T_0 : \frac{C(E_1)}{E_1 \xrightarrow{c} E_2}$$

meaning that the expression on primitives $E_1$ will evolve in the expression on primitives $E_2$ if the condition $C$ on $E_1$ holds. In this case, the evolution is represented by the relation $\xrightarrow{c}$ labeled by the computational cost the system pays for applying such a transformation, that is equal to $c$.

Further, evolving $E_1$ into $E_2$ also means establishing a functional equivalence between them. Two primitives are functional equivalent if they denote the same function when taking the same values as input. Hence, a chain of transformations

$$C_1 \xrightarrow{c_1} E_1 \xrightarrow{c_2} \ldots \xrightarrow{c_n} E_n = \{y_1, \ldots, y_m\}$$

where $E_n$ denotes the final result of evaluating $C_1$, means that in order to get such result, $C_1$ evolves in $n$ functional equivalent (composition of) primitives.

In addition, since each rewriting step $i$ is label with a computation cost $c_i$, a cost for the whole computation can be given as the sum of all the partial costs involved in the evaluation process. As a consequence, each primitive (or composition of primitives) can be associated with a particular computational cost. Moreover, given two primitives $C_1$ and $C_2$ we are able to assert if they are equivalent, thus if they denote the same function but different graphs of control. Provided that $C_1$ and $C_2$ are equivalent and that we are able to associated a computational cost to each of them, we can compare the two computational cost in order to establish which graph is the cheapest one.

In order to prove the feasibility of the approach, we have developed a Java prototype implementation of the programming model. Currently, a package exposing a set of abstract data type, view, access pattern and iterator implementations is provided for programming data parallel concerns. A separate packages collects the set of primitives the user can select to build its application graph and to evaluate it by coupling the appropriate evaluation instance. As shown in the next chapter, the implementation framework has demonstrated the potentiality of the approach both in terms of performance and in terms of expressive power.

Generally speaking, a user program will have a triple structure: in the first part data access is managed by instancing the input dataset (of type `DataContent`) and by applying the appropriate views and iterators. The second part regards the composition of primitives, some of which require as input argument a `Context` object, i.e., an object referring encapsulating some information about the framework runtime.

The third part of the program is represented by the evaluation of the graph coupled with the iterator instantiated on the given input view.

The following code regards an application whose graph is rooted in an `Apply` primitive applying a certain function $f$ (embedded by a `Fun` instance) to all the elements of an input dataset. Such elements are provided by a parallel iterator encapsulating all the data parallel concerns. In particular, it will access all the elements of the input dataset (filtered through an array view) in parallel, thus returning them as a whole.

The method `eval` of the `Evaluator` instance represents the evaluation process by which a graph of primitives coupled with a certain iterator is transformed in the final result by applying a chain of transformations.

```java
// defines a raw set of N values
DataContent s = new DataContentSet(N);

// defines a view on s
ArrayView av = new ArrayView(s);
```
// defines an access pattern i.e. the minimal unit of access of grain 1
ArrayPattern ap = new ByArrayPattern(1);

// defines a parallel iterator on the input view
Iterator it = av.getParIterator(ap);

// sets up the run time environment
Context ctx = new Context();

// defines the graph of primitives
Fun f =...
Apply apply = new Apply(f,ctx);

// evaluates the graph through an Evaluator object
Evaluator evals = ctx.getEvaluatorInstance();
res = evals.eval(it, apply);
1.3 Thesis plan

The thesis is organized as follows.

Chapter 2 presents the technical background we have based our research onto. In particular, we started from coordination models and languages able to keep separated computation from coordination, and following different approaches to parallel programming (data-driven, control-driven and hybrid ones) we point out the need of a programming model with precise characteristic of orthogonalization and semantic description.

Chapter 3 gives an overview of the programming model: in particular, it focuses on the abstract mechanisms and on how they allow separation of concerns. Moreover, it gives a first idea of the semantic model.

Chapter 4 provides the formalization of data parallel concerns: in particular, abstract data types, views, access patterns and iterators are formalized.

Chapter 5 introduces the formalization of control parallel concerns: in particular, it provides a description of the type system to which control primitives belong.

Chapter 6 provides for each primitive the set of inference rules representing the evaluation process. Consequently, it shows how each primitive is evaluated, thus how it can be transformed step-by-step in order to obtain the final application result.

Chapter 7 rewrites the inference rules provided in the previous chapter by labeling them with computational costs in order to provide for each primitive an estimation of the evaluation process.

Chapter 8 shows, through a set of case study, how the system of inference rules can be used to i) prove the functional equivalence between application graphs; ii) select the more efficient graph by comparing the computational costs that can be derived by applying the cost formula related to each component primitive.

Chapter 9 presents the prototype implementation we have developed step by step with the semantic framework. In particular, we will give details on the package implementing data abstractions and the package implementing the set of control primitives and the evaluator function. At the end of the chapter we will show how easily the implementation can be extended by introducing new data abstractions as well as new primitives and we will provide some development guidelines for a fast program development. The chapter is closed by a set of examples of the encoding of some very general applications.

Chapter 10 shows the qualitative results related to the implementation of some sample applications. First, the chapter presents performance measures as completion time, scalability and efficiency of each application. Next, some couples of functional equivalent graphs presented in chapter 8 are analyzed. For each couple of graphs, the theoretical inequality on computational costs predicted in chapter 8 is compared with the one exploited by actual executions. As we will see, all case studies will result right predicted by the cost system. At the end of the chapter, a comparison between theoretical cost and actual cost will be given for some of the applications analyzed in chapter 8.

Finally, we will analyze and discuss the implementation of an application related to the genetic programming field in order to test our approach respect to a more complex, realistic application.

Chapter 11 traces conclusions and future work.
Chapter 2

Background

The first goal of our research work is to keep data parallel and control parallel concerns orthogonal. Our research path started from coordination model and languages because they seemed able to provide the right concepts for abstracting on data and on control. In this chapter we will give a definition of coordination models and languages, an overview of their classification and we will explain how and why they can help us in reaching our goals.

2.1 Coordination models and languages

Over the years, a number of models and metaphors were devised, their purpose being (partially) to abstract away and encapsulate the details of communication and cooperation between a number of entities forming some computation from the actual computational activities performed by these entities.

The coordination paradigm offers a promising way to alleviate this problem and address some of the issues related to the development of complex distributed and parallel systems as these ones were outlined in chapter 1.

The coordination paradigm The coordination paradigm considers programming a distributed or parallel system as the combination of two distinct activities: the actual computing part comprising a number of processes involved in manipulating data and a coordination part responsible for the communication and cooperation between the processes. Thus[THLP98]

\[
\text{Programming} = \text{Computation} + \text{Coordination}
\]

Example 2.1.1 Let A be a process that after an initialization phase, sends an integer value \( m \) to B; B, after its initialization phase, waits for \( m \), increases \( m \) and sends it back to A.

\[
\begin{align*}
\text{Process A:} & : \\
& \text{inizialization\_phase();}
& m = \text{init\_value();}
& \text{send}(B, m);
& \text{other\_computations();}
& \text{receive}(B, m);
& \text{other\_computations();}
\end{align*}
\]

\[
\begin{align*}
\text{Process B:} & : \\
& \text{inizialization\_phase();}
& \text{receive}(A, m);
& m = m + 1;
& \text{send}(A, m);
& \text{other\_computations();}
\end{align*}
\]

Looking at the code for A or/and B it is evident that the communication statements (such as sending and receiving primitives) are interleaved with computational ones, represented here by the \text{other\_computations()} and \text{inizialization\_phase()} calls and by the \text{add (+)} instruction.
The separation between communication and computation activities is not always so clear and well bounded and programmers generally write their code thinking at the same time both about how data is computed and how the several parts of an application have to interact with each other to share or communicate data. Such mental effort makes writing parallel and distributed code a non-trivial task. In short, coordination may help to distinguish the computational concerns of some distributed or parallel applications from the communication ones, allowing the separate development as well as the eventual amalgamation of these two major development aspects.

In the example given above, the concept of coordination will keep separated the aspect related to the communication between A and B from the computational ones. So, it will be possible to express the communication graph of the application by means of a number of ad hoc coordination directives embedded into a host language as a library routine. The programmer defines the implementation parts of his application by means of the host language and composes them into a graph of processes using the coordination language. This is an example of the correct way to reason about the coordination paradigm fashion.

**A formal definition**  The concept of coordination plays a key role in many diverse disciplines such as economics and operational research, organization theory and biology. Consequently, there are many formal definitions of what coordination is. In the area of Programming Languages, probably the most widely accepted definition is given by Carriero and Gelernter in [GC92]:

> **Coordination is the process of building programs by gluing together active pieces**

Consequently

A coordination model is the glue that binds separate activities into and ensemble.

A coordination model could be formalized as a triple

\[
< E, L, M >
\]

where \( E \) represents the entities being coordinated, \( L \) the media used to coordinate the entities and \( M \) the semantic framework the model adheres to.

From our point of view, the concept of coordination is related to the orthogonalization of data and control concerns. Thus, coordinating should mean providing a separate and independent description of data parallelism with respect to control parallelism in order to get all the advantages we listed in chapter 1.

We argue that in our design for orthogonality between data and control parallel concerns, \( E \) will be populated by both data access/handling and control directives. \( L \) will be represented by some abstractions able to combine the elements populating \( E \). \( M \) will be the semantic framework in which \( E \) entities and \( L \) media will be formalized together with some provable properties on them.

Coordination is closely related to those of multilinguality and heterogeneity. Since the coordination component is separate from the computational one, the former views the processes comprising the latter as black boxes; hence, the actual programming language used to write computational code plays no important role in setting up the coordination apparatus. Moreover, since the coordination component offers a homogeneous way for interprocess communication and abstracts away the machine-dependent details, coordination encourages the use of heterogeneous ensembles of architectures.

With respect to the problem of writing massively parallel applications it must be noticed that the models of cooperation used in today’s concurrent applications are essentially a set of ad hoc templates that have been found to be useful in practice. There is no paradigm wherein we can systematically talk about cooperation of active entities, and wherein we can compose cooperation scenarios such as (and as alternatives to) models like client-server, worker pools, etc., out of a set of primitives and structuring constructs in new or old languages. Consequently, programmers
must directly deal with lower-level communication primitives that comprise the realization of the cooperation model of a concurrent application.

The inability to deal with the cooperation model of a concurrent application in an explicit form contributes to the difficulty of developing working concurrent applications that contain large number of active entities with non-trivial cooperation protocols. In spite of the fact that the implementation of a complex protocol is often the most difficult and error prone part of an application development effort, the final result is typically not recognized as a “commodity” in its own right, because the protocol is only implicit in the behavior of the rest of the concurrent software. As a consequence, maintenance and modification of the cooperation protocols of concurrent applications are much more difficult than necessary, and their reuse next to impossible.

**Coordination languages** From a merely linguistic point of view,

*A coordination language is the linguistic embedding of a coordination model, offering facilities for controlling synchronisation, communication, creation and termination of computational activities.*

Thus, if we adopts a slightly liberal view of what coordination is, we can also include configuration and architectural description languages in the category of coordination languages. Reversely, we can also view coordination as dealing with architectures (i.e. configuration of computational entities).

A coordination language can be thought as the linguistic counterpart of a number of software platforms and libraries presently popular for easy development of concurrent applications. Such systems, e.g. PVM [PVM], MPI [MPI], CORBA [Cor], etc. are sometimes called *middleware*: typically they are built as a number of software layers between communicating processes (client and server) that deliver extra functionality. In particular, they hide the complexity of coordination issues behind a common set of APIs that processes can invoke.

A significant number of these models and languages are based on a few common notions, such as pattern-based and associative communication, to complement the name-oriented, data-based communication of traditional languages for parallel programming [WA99]. Coordination languages have been also applied to the parallelization of computation intensive sequential programs in the fields of simulation of Fluid Dynamics systems, matching of DNA strings, molecular synthesis, parallel and distributed simulation, monitoring of medical data, computer graphics, analysis of financial data integrated into decision support systems, and game playing (chess).

In section 2.2 we will give an overview of how coordination models have been classified in the literature starting from its two extreme exploitations, *data-driven* and *control-driven* approaches. We will prove in section 2.3 that such a classification perfectly matches data parallel and control parallel separation, also comparing different models and/or languages provided in literature. Finally, section 2.4 will discuss all the still open problems related to the coordinate exploitation of data and control parallelism.

### 2.2 Classification

There are a number of dimensions in which one can classify coordination models and languages, such as the kind of entities that are being coordinated, the underlying architectures assumed by the model, the semantics a model adheres to, issues of scalability, oneness, etc.[CH96, ACH98].

Coordination models and languages can be classified as either *endogenous* or *exogenous* [Arb98]. Endogenous models and languages provide primitives that must be incorporated *within* a computation for its coordination. In applications that use such models, primitives that affect the coordination of each module are inside the module itself. In contrast, exogenous models and languages provide primitives that support coordination of entities by means of *external* mechanisms.
In applications that use exogenous models, primitives that affect the coordination of each module are outside the module itself.

Endogenous models are sometimes more natural for a given application. However, they generally lead to a mixture of coordination primitives with computation code, which entangles the semantics of computation with coordination protocols. This intermixing tends to scatter communication/coordination primitives throughout the source code, making the cooperation model and the coordination protocol of an application nebulous and implicit: generally, there is no piece of source code identifiable as the cooperation model or the coordination protocol of an application, that can be designed, developed, debugged, maintained, and reused, in isolation from the rest of the application code. This is exactly the situation represented in the piece of code of the example given in section 2.1.

On the other hand, exogenous models encourage development of coordination modules separately and independently of the computation modules they are supposed to coordinate. Consequently, the result of the substantial effort invested in the design and development of the coordination components of an application can manifest itself as tangible “pure coordination modules” which are easier to understand and can also be reused in other applications.

Anywhere, all the models fall into one of the two major categories of coordination programming, namely either data-driven or control-driven (or task-driven or process-oriented).

2.2.1 The data-driven model

The main characteristic of the data-driven coordination models and languages is the fact that the state of the computation at any moment in time is defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components. In other words, a coordinator or coordinated process is responsible for both examining and manipulating data as well as for coordinating either itself and/or other processes by invoking the coordination mechanisms each language provides.

Almost all coordination models belonging to the data-driven category have evolved around the notion of Shared Dataspace. A Shared Dataspace is a common, content-addressable data structure. All processes involved in some computation can communicate among themselves only indirectly via this medium. Since interprocess communication is done only via the Shared Dataspace and the medium’s contents are independent of the life history of the processes involved, this metaphor achieves the decoupling of processes in both space and time.

Some processes can send their data into the medium and then carry on doing other things or even terminate execution while other processes asynchronously retrieve this data; a producer need not know the identity of a consumer (and vice versa) or, indeed, if the data it has posted into the medium has been retrieved or read by anyone.

Example 2.2.1 Figure 2.1 gives an ideal representation of a shared dataspace: round objects represent processes whereas square ones represent data. Empty square boxes are templates and are used by processes to somehow specify what sort of data should be retrieved from the shared medium. Filled square boxes represent various kinds of data structures. Finally, small round objects represent “active” data structures; in reality, these are processes which after their execution ends, turn into passive ordinary data. A process may be a pure producer or a pure consumer or both.

The various models in this category are different with respect to a number of parameters. For instance, one parameter is the actual structure of data; in some cases they are flat tuples or records and in other cases nested tuples and records are also supported. Another parameter is the actual mechanism employed to retrieve data. Some models are based on various forms of pattern matching, whereas others use more sophisticated techniques that view the shared medium as something more than a flat unstructured space (e.g. they view it as a multiset). Yet another distinction is related to issues such as locality of reference within the shared medium, security and efficiency.
2.2. CLASSIFICATION

Data-driven coordination languages are typically endogenous systems. Focusing the description of a program behavior on how data are accessed, their more natural exploitation is given by a set of primitives embedded in a host language that describes explicitly the handling of data. Thus, such coordination primitives (coupled with a coordination metaphor) are mixed within the purely computational part of the code. They encapsulate in a useful way the communication and configurational aspects of some computation, but must be used in conjunction with the purely computational manipulation of data associated with some process. So, processes cannot easily be distinguished as either coordination or computational ones. This doesn’t mean that there does not exist a useful and clear separation between the coordination functionality and the purely computational functionality of a process. But it usually means that, at least stylistically and linguistically, there exists a mixture of coordination and computation code within a process definition. It is usually up to the programmer to design his program in such a way that the coordination and the computational concerns are clearly separated and are made the responsibility of different processes; however, most of the time such a clear separation is not enforced at the syntactic level by the coordination model.

Strictly speaking, not all coordination models in the data-driven category follow the above pattern of coordination. For instance, some tools based on the Actor Model [Agh90] of computation use a message-passing (rather than a Shared Dataspase) base mechanism.

2.2.2 The control-driven model

In the control-driven coordination model the state of the computation at any moment is only defined in terms of the coordinated patterns that the processes involved in some computation adhere to and the coordinated framework evolves by means of observing state changes in processes and, possibly, broadcast of events. The actual values of the data being manipulated by the processes are almost never involved. Processes are treated as black boxes. Processes communicate with their environment by means of clearly defined interfaces, usually referred to as input or output ports. Producer-consumer relationships are formed by means of setting up stream or channel connections.
between output ports of producers and input ports of consumers. By nature, this connection are point-to-point, although limited broadcasting functionality is allowed.

In addition to using ports, processes often send out to their environment control messages or events with the purpose of letting other interested processes know in which state they are or informing them of any state changes.

![Control-driven coordination model](image)

**Figure 2.2: Control-driven coordination model**

**Example 2.2.2** Fig.2.2 depicts these concepts. The figure shows a configuration involving one producer with one input and two output ports and two consumers, one with a single input port and a single output port and the other with two input ports and one output port. Stream connections have been established between the output ports of the producer and the input ports of the consumers, sometimes with more than one stream entering an input port or leaving an output port. Furthermore, the producer and one of the consumers either raises and/or observes the presence of some events.

Also in the case of control-driven model there are several variants. For instance, in some languages events can be parametric with types and data values (effectively another mechanism for interprocess communication) whereas in other languages events are strictly simple units signifying state changes. Furthermore, in some languages events are broadcasted by means of mechanism different from stream connections whereas in other languages events actually are passed through streams. Stream connections themselves can be realised in a number of ways; for instance, they may or may not support synchronous communication. In some cases streams are interfaced to some common medium (such as “data bus”) rather than being point-to-point connections between ports; even in this latter case however, the medium is not used for unrestricted broadcasting. Also, some languages support dynamic creation of ports and exporting of their id for use by other processes whereas others limit such functionality.

Stylistically the coordination component is almost completely separated from the computational ones; this is naturally achieved by defining a brand new coordination language where the computational parts are black boxes. For this reason control-driven coordination models can be inserted in the class of exogenous models. Computational parts are somewhat combined, composed to each other with an “external” instrument that is the coordination language.

Consequently, whereas in the case of the data-driven category, the coordination component is usually a set of primitives with predefined functionality which is used in connection with some host computational language, in the control-driven category the coordination component is usually a fully-fledged language. This also means that it is easier in the second case (in fact, it is usually being enforced by the model) to syntactically separate the processes (or at least program modules) into two distinct groups, namely purely computational ones and purely coordination ones.
2.3 Coordination models for orthogonal data and control parallelism handling

Aside from the stylistic differences between the two categories which affect the degree of separation between computational and communication concerns, each of the two categories also seems to be suitable for a different type of application domain. The data-driven category tends to be used mostly for parallelising computational (thus data intensive) problems. The control-driven category tends to be used primarily for modeling systems, thus in all those cases in which communications and synchronizations are the most critical part of the application. Such a difference may be attributed to the fact that from within a configuration component a programmer has more control over the manipulated data in the case of data-driven coordination languages, than in the case of control-driven ones. Thus, the former category tends to coordinate data whereas the latter tends to coordinate entities (which, in addition to ordinary computational processes, can also be devices, system components, etc.). However, it must be clear that this trend is not for a definitive separation of application domains. In fact, there are data-driven coordination languages used for distributed systems and control-driven coordination languages used for parallelizing data-intensive programs.

In the following sections we will give an overview of several approaches based on data-driven, control-driven or hybrid coordination models and for each of them we will briefly discuss how they handle data and control parallelism. At the end of the section it will be clear how data-driven based approaches better exploit data parallelism, while control-driven based approaches fit more likely control-parallel concerns. We present the works available dividing them in the three main categories, referring them as approaches just because some of them are mechanisms for parallel programming, missing a real programming model.

2.3.1 Data-driven approaches

As largerly seen above, the activity in a data-oriented application tends to center around a substantial shared body of data; the application is essentially concerned with what happens to data.

**Linda** [CG89b, ACG94] is historically the first genuine member of the family of coordination languages. It provides a simple and elegant way of separating computation from communication concerns. Linda is based on the so-called *generative communication* paradigm: if two processes wish to exchange some data, then the sender generates a new data object (referred to as a *tuple*) and places it in some shared dataspace (known as a *tuple space*) from which the receiver can retrieve it. This paradigm decouples processes in both space and time: no process need to know the identity of other processes, nor is it required of all processes involved in some computation to be alive at the same time. In addition to passive tuples containing data, the tuple space can also contain active tuples representing processes which after the completion of their execution, turn into ordinary passive tuple.

Tuples are actually sequences of typed fields. They are retrieved from tuple space by means of *associative pattern matching*. More to the point, the parameter of the primitives is actually a *tuple schemata* containing formal parameters; pattern matching of the parameter $\tau$ with an actual tuple $ta$ in the tuple space will succeed provided that the number, position and types of $\tau$’s fields match those of $ta$.

Linda is not a fully flagged coordination language but a set of some simple coordination primitives that express read/write operations on tuples.

Linda primitives are completely independent from the host language; thus, it is possible to derive natural Linda variants of almost any programming language or paradigm (imperative, logic, functional, object-oriented, etc.). Linda’s “friends” are C, Modula, Pascal, Ada, Prolog, Lisp, Eiffel and Java to name but a few [ACG86].
Nevertheless, Linda provides an endogenous model exploiting all the limitations of having a set of coordination primitives that must be intermixed with computational code. For these reasons, its approach can be thought as a plain data-parallel programming system where the orthogonality is rather a logical one, completely missed in terms of written code.

**High Performance Fortran** (HPF) [For97] is a set of Fortran extensions primarily oriented towards the specification of data parallel algorithms. It provides annotations for the distribution and alignment of data, and for the specification of parallel loops. The discussion of the HPF programming model has to take into account two levels. At the higher level, the execution of an HPF program is modeled by a single process. Even though users directly provides the data mapping through directives and explicitly express data parallelism via array statements, forall constructs and the independent directives, the model stresses the important point that HPF programs operate in one global data space.

The second lower level reflects an execution model that is part of the HPF language definition and follows the SPMD paradigm for distributed memory machines. It is based upon a set of abstract processors with disjoint address spaces, each of which executes exactly one process. Data distribution directives partition the (global) address space of the first layer across the abstract processors of the second layer; the on directive performs the analogous job for parallel execution control and work sharing.

HPF fully supports those aspects of multidisciplinary applications (MDAs) that are related to the execution of data parallel programs. In addition, it also offers simple tasking features, which allow the specification of parallel sections, executing on disjoint set of processors, without a mechanism for explicit synchronization or communication. Thus, in this sense, HPF can be considered a coordination language but there are many dialects of it that better support orthogonality. In fact, the rudimentary nature of the HPF instruments, makes them not suitable for expressing the complex interaction among asynchronous tasks as required by the family of MDAs. Similarly, even though the module construct for Fortran provides a limited modularization and encapsulation facility, it does not sufficiently meet the demands of MDAs.

The concept of compositionality [DGT93] shares the same goals with coordination, namely reusability of sequential code, generality, heterogeneity and portability; as such it can be seen as a coordination model. A compositional programming system is one in which proprieties of program components are preserved when those components are composed in parallel with other program components. Thus, it is possible to define in a compositional way recurring patterns of parallel computation, whether configuration ones (such as mapping techniques) or communication ones (such as streamers and mergers) as building blocks and combine them together to form bigger programs. If desired, the compositional assembly preserves the deterministic behaviour of its constituent parts, thus simplifying program development by allowing program components to be constructed and tested in isolation from the rest of their environment.

There are basically two approaches to deriving compositional programs. The first approach is based on concurrent logic programming and is exemplified by related languages. The second approach for deriving compositional programs originates from functional programming and is expressed in the form of skeletons. Skeletons [Col89, Só9b, DGTY95, Ske, DFH+93] are higher order functional forms with built-in parallel behaviour. They can be used to abstract away from all aspects of a program’s behaviour and inherit all the desirable properties of the functional paradigm such as abstraction, modularity and trasformation. The latter capability allows a skeletons-based program to be transformed to another, more efficient one, while at the same time preserving the properties of the original version [ACD98, DGTY95]. Thus, all analysis and optimizations can be confined to the functional coordination level which is more suitable for this purpose. Furthermore, one is able to reason about the correctness of the programs produced or derived after transformations [AD99, Dan99, MM02]. Skeletons can be both configuration and computational ones and, being independent of the host computational language, they can be combined with C, Fortran, etc. [DS01, SGCD01, Só1b, Só1a, Só9a]
There is no unique set of skeletons and a number of them have been designed with some special purpose in mind. At first, the skeleton approach was per se a data parallel one since the state of the computation was given at each time by the state of the data flowing through the application components. Nowadays, as we will see in section 2.3.3, the evolution research in this field has produced a number of solutions leading to hybrid skeleton approaches aiming at mixing data and task parallel concerns.

Opus [MRZ98] is effectively a coordination superlanguage on top of HPF for which it was designed, for the purpose of coordinating concurrent execution of several data parallel components. Interaction between concurrently executing tasks is achieved via the Shared Data Abstraction (SDA), a Linda-like common forum. An SDA is in fact an Abstract Data Type containing a set of data structures that define their state and a set of methods for manipulating this state. SDAs can be used either as traditional ADTs that act as data servers between concurrently executing tasks or as computation servers driven by a main, controlling tasks. In that respect, Opus combines data and task parallelism.

Execution of an Opus program begins with a single coordinating task that establishes all the participating computation and data servers. The coordinating task drives the computation by invoking the proper methods within the computation SDAs. Communication and synchronisation between the concurrently executing tasks is managed by the data SDAs.

Opus is one of the few language in the data-driven category which separates quite clearly the coordination component from the HPF computational component. However, I choose to include the model in this category because of the SDA mechanism it employs which is quite reminiscent to the Shared Dataspace one.

2.3.2 Control-driven approaches

The activity in a control-oriented application tends to center around processing or flow of control and, often, the very notion of data, as such, simply does not exist; such an application is essentially described as a collection of activities that genuinely consume their input data, and subsequently produce, remember and transform “new data” that they generate by themselves.

The Message Passing Interface (MPI) [MPI] assumes that a fixed set of processes is (implicitly) created at program initialization, each executing exactly one process. There is no explicit mechanism for process creation.

MPI provides several functions, most with a number of variants, for interprocess communication. In addition to point-to-point communication, these include collective communication facilities covering barriers, broadcast, gather, scatter and reduction operations. The scope of a collective operation can be restricted using communicators, which define a group of processes and a communication context. MPI 2.0 supports process creation and management, one-sided communications, extended collective operations, external interfaces, I/O and additional language bindings.

MPI provides some supports for MDAs. In particular, the modularity of the MPI programming model can be exploited to map individual discipline codes to separate modules, with only limited recoding necessary in many cases. On the other hand, explicit message passing represents a very low level for expressing inter-process communication. Furthermore, there is no explicit support for data parallelism so that all the details related to partitioning and distribution of data sets must be treated explicitly by the programmer.

MANIFOLD [PA97, Arb96, AHS93] is one of the latest developments in the evolution of control-driven or process-oriented coordination languages. As is the case in most of the other members of this family, MANIFOLD coordinators are clearly distinguished from computational processes which can be written in any conventional programming language augmented with some communication primitives. Manifolds (as MANIFOLD coordinators are called) communicate by means of input/output ports, connected between themselves by means of streams. Evolution of a
MANIFOLD coordination topology is event-driven based on state transitions. More to the point, a MANIFOLD coordinator process is at any moment in time in a certain state where typically it has set up a network of coordinated processes communicating by sending and/or receiving data via stream connections established between respective input/output ports. Upon observing the raising of some event, the process in question breaks off the stream connections and evolves to some other predefined state where a different network of coordinated processes is set up. Note that, unlike the case with other coordination languages featuring events, MANIFOLD events are not parameterised and cannot be used to carry data - they are used purely for triggering state changes and causing the evolution of the coordinated apparatus.

2.3.3 Hybrid approaches

In the sequel I describe a number of systems that integrate task and data parallelism in different ways. I recognize that there are other systems which integrate task and data parallelism but this is not intended to be an extensive survey of the field. Rather, I selected two languages with task parallel background and two languages with a data parallel background and then an overview of other alternative approaches is given.

Orca task parallel language [HBJ98, HB96] has been extended with constructs for data parallelism, resulting in a language with mixed parallelism. Orca supports general task parallelism and allows dynamic creation of processes and mapping to processors. Communication in Orca is based on shared object, which are instances of Abstract Data Types (ADTs). Processes in Orca communicate by applying user-defined ADT operations on shared objects. Such an operation can be applied to only a single object and is always indivisibly executed.

In the extended Orca language, data parallelism is expressed through partitioned objects, which are objects containing arrays that can be distributed among multiple processors. A data parallel operation on a partitioned object is executed in parallel by these processors; each processor applies the operation to the element it “owns” (the owner computes rule). The distribution of the elements is expressed by the user, by invoking a runtime primitive that specifies the set of processors to use and the owner of each element. This distribution may be changed during runtime. The programmer can also cluster elements that are accessed together into so-called partitions. Whenever an operation needs an element from a remote processor, it will actually fetch the entire partition the element belongs to, thus increasing the granularity of the data transfers.

Task and data parallelism can be used together in a single program and in a clean and simple way, using a general object model that supports both replicated and partitioned objects. The model, however is biased on task parallelism. It supports general task parallelism (e.g. dynamic process creation) but imposes several restrictions on data parallelism. Programs are required to adhere to the owner-compute rule, which may not always be appropriate.

In conclusion, the extended Orca language gives the full power of task parallelism with some of the functionality of data parallelism, integrated in a clean model. It does not have the full power of data parallel languages like HPF which also is reflected in the usage of extensive runtime support and little compile support.

Braid [WG94] is a data parallel extension to the Mentat Programming Language (MPL). MPL is an object-oriented task parallel language based on C++. The MPL programmer can designate certain classes as Mentat classes. Operations on objects of Mentat classes are executed in parallel, using a macro data-flow model.

Braid logically extends this model by introducing data parallel Mentat classes. Objects of such classes are partitioned among multiple processors. Operations on these objects are executed in a data parallel way, much as in Orca.

A novel idea in Braid is subset level data parallelism, which can be used to define operations on entire subsets of an object’s data.

Braid allows operations to take data parallel objects as parameters.
The distribution strategy for an object is determined by the compiler and runtime system, using annotations provided by the programmer. These annotations describe the communication behavior of objects. The simplest annotation expresses the local behavior within one object. It can indicate, for example, that an operation on an element usually also accesses the four neighbors for the element. Another annotation specifies nonlocal communication behavior between objects. Other annotations exist to specify which classes of objects often interact and which operations are the dominant ones.

Another work that integrates coordination features is presented in [FKKC97] and it is based on specialized coordination libraries designed to be called from data-parallel programs. These libraries support an execution model in which disjoint process groups (corresponding to data-parallel tasks) interact with each other by calling group-oriented communication functions. In keeping with the sequential reading normally associated with data-parallel programs, each task can be read as a sequential program that calls equivalent single-threaded coordination libraries. The potentially complex communication and synchronization operations required to transfer data among process groups are encapsulated within the coordination library implementations.

At the moment, to illustrate and explore this approach, it has been defined and implemented a library that allows the use of a subset of the Message Passing Interface to coordinate HPF tasks. It defines functions for both point-to-point and collective communication among tasks executing in separate address spaces; its definition permits efficient implementations on both shared and distributed-memory computers.

P3L, the Pisa Parallel Programming Language [BDP94, DOP91, DMO\textsuperscript{+}94], is a structured, parallel programming language suitable for programming massively parallel MIMD machines. This language comes with a powerful set of compiling tools that solve most of the problems that usually are in charge to the programmer (such as mapping and load balancing), by adopting a template-based compiling strategy. The form of parallelism realized by the implementation templates mirror those present in the source language under the form of parallel constructs. A parallel construct is a typical pattern of parallel computation made available to the programmer under the form of a primitive statement of the language.
P3L includes task parallel skeletons and data parallel skeletons[DPP97] and they can be combined on the basis of the two-tier model. In general, a computation consists of nested task parallel constructs where atomic task parallel computation may be sequential or data parallel. Purely data parallel and purely task parallel computations are special cases of this model.

The programming environment SkIE (Skeleton-based Integrated Environment) [BDPV99], developed inside the PQE2000 project, includes a coordination language, SkIEcl, allowing the designers to express, in a primitive and structured way, efficient combinations of data parallelism and task parallelism. Modules developed with standard languages and tools are encapsulated into SkIEcl structures to form the global application. Performance models associated to the coordination language allow powerful optimizations to be introduced both at run time and at compile time without the direct intervention of the programmer.

The skeleton approach indicated above as an example of compositional programming has been studied to deal with mixing task and data parallel application in spite of its intrinsically data parallel nature. Many works have been done to make this approach feasible as a hybrid model and many of them have lead to the development of libraries [DT02] able to support both parallel computation, typically by means of a run-time support based on a two-tier model architecture taken from P3L.

One of the most recent implementation is the work by H. Kuchen. He has shown [Kuc02, KC02] that it is possible to provide algorithmic skeletons in form of a library rather than within a new programming language. This will facilitate their use for typical parallel programmers. The library smoothly combines the main features of existing skeletons. In particular, it provides task parallel
skeletons generating a system of communicating processes as well as data parallel skeletons working in parallel on a distributed data structure. Task and data parallelism are combined based on the two-tier model and the notion of distributed data structure. This data structure is manipulated by operations which process it as a whole and which happen to be implemented in parallel internally. These operations can be interleaved with sequential computations working on non-distributed data. In fact, the programmer views the computation as a sequence of parallel operations. Communication consists of the exchange of the partitions of a distributed data structure between all processors participating in the data parallel computation.

Kuchen’s library was born from previous works on Skil [BK96c, BK98, BK96a, BK96b, GHB96, BK96c, BK98], acronym for Skeleton Imperative Language, in which the notion of distributed data structure was introduced for the first time.

An alternative solution is provided by [SGD99], in which skeletons are provided inside a functional environment. Each skeleton is represented by a higher order function that can be combined with other sequential functions or skeleton. Since the project is particularly addressed to vision applications, data is often represented as a flow of values (representing images) on which complex functions can be applied. The attention is particularly focused on the exploitation of control parallelism although some typical data parallel skeletons are provided.

In [Col04, BC05] a C/MPI programmer interface with a collection of collective operations representing recurring pattern of computation and communication is presented. The processes participating to a call are grouped according to the semantics of the represented skeleton and the interaction among processes belonging to same group are facilitated by communication routines effecting only their own context. For this reason, the model on which such library is based, has been defined a “process based model”. On the other hand, data access need to be manually handled within the address space of each process by following the C/MPI syntax conventions, although the library offers some specific interaction functions for a safer structuring and exchanging of data between processes.

In [DRST01] another way of integrating task and data parallelism using skeletons is presented. This approach, which is called DIP (Domain Interaction Patterns) is a high level coordination language to express task parallelism among a collection of data parallel HPF tasks, which interact according to static and predictable patterns. DIP allows an application to be organized as a combination of common skeletons, such as multi-blocking or pipelining. Skeletons specify the interaction among domains involved in the application, along with the mapping of processors and data distribution.

Another alternative approach is given by the family of parallel frameworks such as CO2P3S [BMA*02, MBS*], based on an object-oriented pattern-based model [MSS99, MSS97]. CO2P3S supports multiple levels of abstraction, allowing the user to design an application with high-level patterns, but move to lower levels of abstraction for performance tuning. Patterns are implemented as parameterized templates, allowing the user the ability to customize the pattern to meet their needs. CO2P3S generates code that is specific to the pattern/parameter combination selected by the user. CO2P3S addresses separation by expressing the parallelism diagrammatically and allowing application-specific code to be inserted into the pattern. In this case, the diagram is a collaborative diagram. Further, the CO2P3S approach of generating code that invokes user hooks, provides more opportunity to re-use the hooks when the user changes the pattern. Since the hook code does not affect the communication flow of the program (because the communication code is generated and cannot be edited by the user), there is a greater possibility that this code can be used in another pattern.

Parallelism in a program can be hierarchically specified by allowing patterns to be substituted for the sequential components in a program. This approach provides a structured way of building complex program elements.

It must be seen that, even if the layered structure of the framework give the possibility to hide communication details and to focus on the computational aspects of the application (hence giving
good opportunity of separation), mixing task and data parallelism is not, at the moment, among the main goal of CO$_2$P$_3$S project. Anywhere CO$_2$P$_3$S is not the only example of framework in which the pattern paradigm is used [MMS01, WS98].

In [RR99] a model for a systematic derivation of efficient message-passing programs with mixed task and data parallelism is presented. It separates the algorithmic properties from the properties of the target machine and partitions the derivation process of a parallel implementation for a given algorithm into several steps. Data parallel executions are expressed as modules that can be executed by a varying number of processors. Depending on their data dependencies, different modules can be executed concurrently to each others on disjoint groups of processors or must be executed consecutively. A specification language provides constructs to express possible execution orders between modules and thus describes the available degree of task parallelism explicitly. The transformation of the specification program into a coordination program describes how the available degree of parallelism is actually exploited for a specific parallel implementation. The result is a complete description of a parallel program that can be easily translated into a message-passing program. The coordination program is expressed in a coordination language which is an extension of the specification language. The final message-passing program is expressed in C with MPI.

ASSIST (A Software development System based upon Integrated Skeleton Technology) [Van02] is a proposal of a new programming environment oriented to the development of parallel and distributed high-performance applications according to a unified approach. The proposal is based on the evolution and joining of the software component technology and of the structured parallel programming technology. ASSIST intends to overcome some limitations of the classical skeletons approach, in particular for complex combinations of task and data parallelism. A new paradigm, called parallel module (parmod), is defined which, in addition to expressing the semantics of several skeletons as particular cases, is able to express more general parallel and distributed program structures. ASSIST allows the programmer to design the applications in the form of generic graphs of parallel components, including both data-flow and nondeterministic reactive computation. Another distinguishing feature is that ASSIST modules are able to utilize external objects, including shared data structures and abstract objects (e.g. CORBA), with standard interfacing mechanisms. In turn, an ASSIST application can be reused and exported as a component for other applications, possibly expressed in different formalisms.

2.4 Discussion

In spite of the variety of languages developed for writing parallel programs in a powerful and expressive manner, no one of them has been able to impose itself among the scientific and industrial community. This means that it is still actual and pressing the need of a model (and, thus of a language) able to support parallel programming in a general-purpose fashion such that industrial, scientific and academic community can overcome the feel that parallel programming is a difficult task and parallel applications can be written efficiently with much less efforts. There are several reasons for this lack:

1. Nowadays it does not exist a unifying model able to support general purpose applications. There exist a number of middleware instruments and tools (such as CORBA[Cor], MPI[MPI], PVM[PVM], etc.) that are able to implement in some (not always simple) way parallel processing but they miss a reference model enforced by a formal semantics able to demonstrate some specific features about correctness, soundness, efficiency and to provide performance proofs.
There are a number of frameworks (e.g. COPS [BMA+02, MBS+, MSS99, MSS97]), libraries (e.g. Kuchen’s [Kuc02], Foster’s [FKKC97], some Linda’s descendants) or parallel constructs of ad hoc languages (P3L [DOP91], SKie [BDPV99], ASSIST [Van02], Manifold [PA97, Arb96, AHS93]) or host languages (e.g. Braid [WG94]) but no one is supported by a clear semantics able to justify the relative effectiveness in writing efficient and general-purpose parallel applications. Every one seems to be borne as experiment attempting to obtain something feasible to produce parallel code with acceptable performance.

2. There does not exist a general pourpose model and language accepted from everyone; instead, many of the tools available focuses on a specific domain application and this make much more difficult the design, the large scale diffusion and the definitive assumption of a unifying model.

Actually, research world is still investigating on which implementation is better to produce such tools [Col04]. In fact, someone argues that a library of primitives to integrate within existent environments is more a feasible and acceptable approach; someone else, instead, believes that ad hoc languages have to be provided, focusing on the hot topics of parallel programming: simple programmability, portability, performance and static dynamic optimizations.

3. One of the main problem in thinking about a model suitable for parallel programming, is often that researchers focus their design on data or (exclusively) on control; this comes from the fact that, as explained in chapter 1 there exist two historical approaches for parallel programming: task parallel and data parallel. All the tools and the environments developed since now take into account this distinction and, overall, they are based on design decisions conditionated by this orthogonal duality. So, as result, we have some instruments that behave in a very efficient way while handling task parallel computation but lacking in data parallel one (and vice versa): the difficulties in such situations are that a control-oriented language (often endogenous) will ask the programmer to specify explicitly the partitioning and the distribution of data because it needs to see the unity of computation as an isolated task; hence, the programmer must supply not only the “pure” computational modules but also “structural” ones in which data are marshalled and unmarshalled (or “packaged” for computation). The insertion of such additional elaboration makes writing parallel code more complex and, hence, less attractive and obviously hacks performance.

On the other hand, using a data parallel language for expressing task parallel computation (or mixed one) is difficult too, because the lower data-driven model doesn’t offer the concept of flow of control as something having its own life cycle; it doesn’t give the notion of stream or channel by which two or more modules can exchange their data because it “thinks” in terms of parallel operations on a (set of) data instead of structures of control throughout messages (anonymous data) are transferred.

This concept will be more clear with an example

4. In the last years, scientific world has attempted to overcome the problems raising from the dual nature of parallel computation, mixing together data parallelism and task parallelism in an ensamble.

The results of this efforts have been, once more, a number of tools and methods to express with the same language (or library or framework) both task and data parallelism.

However, this “ibrid” solutions, together with those pointed in the first item, fail again since they miss a clear reference semantics, a concrete model able to describe parallel computation in a formal and verifiable way.

This lack comes from two reasons: the first is that, another time, ibrid languages was born as experimental attempts to have an acceptable performance or something useful “on the fly”. The second reason is that, in reality, every ibrid model is based on a two-tier vision of computation: we have pure task-parallel languages enriched with data-parallel constructs that hide details on partitioning and distribution of data to the programmer; or we have pure
data-parallel languages enriched with task-parallel constructs (or control flow constructs) that allow the programmer to specify some forms of process control.

That’s not enough. Mixing data and task parallelism in this fashion means anyway forcing the expression of a certain type of parallelism. We have simply make up the data or task parallel nature of a problem with task or data parallel embodiments.

In some sense, the ibrid models today available have attempted to give a unifying language for data and task parallelism, enforcing the difference between the two type of computation through their two-tier vision.

My opinion is that the difference must be stressed but not in a vertical sense but in orthogonal sense. What this means will be clearer in the following sections.

5. the previous point evidences that any of the languages today enable doesn’t achieve a concrete decoupling between control flow and access to data. This means that programmers, also in the ibrid model, are forced to program thinking at the same time about the computational and the control concerns. Such cooperation makes parallel programming less attractive and more error-prone both to programmers and to the industrial world.

6. Many efforts have devoted to produce frameworks, libraries or parallel programming languages based on conventional programming languages that are basically high level, complex version of the von Neumann computer [Bac78].

Because of the wide diffusion of the Von Neumann programming style it’s difficult to overcome the view of sequential programming that forces thinking in word-at-a-time terms. On the basis of sequential programming there is the assignment statement responsible for altering variables that are imitations of computer’s storage cells.

Programming parallel applications is a very different kind of mental activity because, after accessing data stored locally, there are several more factors to take into account. From this point of view the approach followed by structured parallel programming methods is better because it has a compositional view of modules, very close to the functional programming philosophy [Bac81] but they accomplish little in attacking the fundamental problems created by the word-at-a-time von Neumann style of programming with its primitive use of loops, subscripts and branching flow control.

It might be better being able to express what are the functions to elaborate, having an intrinsic mechanism that “knows” what can be done in parallel and what doesn’t.

7. Another aspect to point out is that it would be convenient to obtain our new model/language without introducing another programming language that programmers should be reluctant to learn. It will be better if we are able to take advantage of all the resources and knowledge on hand at the moment.

Observations as the previous ones have encouraged us to think about an alternative solution to model and write parallel applications, aiming at overcoming the weakness of two-tier models though new type of abstractions. In addition, we tend to the design of a model for parallel programming strongly guaranteed by a formal semantics allowing us to establish and prove some properties on the program structures and to reason independently in terms of control, data accesses and, finally, in terms of coordination (orthogonalization). Moreover, we aim at designing a semantic framework general enough to cover a high number of applications and, in the meanwhile, we aim at providing an implementation to test the feasibility of such an approach.
Chapter 3

The programming model: an overview

The first goal of this thesis is to design a programming model based on a clear distinction between data and control parallelism exploitation mechanisms. In fact, by keeping data and control parallelism separated, we would achieve a perfect degree of orthogonalization that can be used both to express data and control parallelism exploitation in some kind of handy way and to allow programmers to design new parallelism exploitation patterns out of the existing, data and control parallel ones. In order to achieve such a separation, we must provide a formalism that allows programmers to describe both kind of parallelism exploitation in an independent way.

Our goal is achieved by providing within the programming model, clearly defined and semantically well founded collective data accesses and control primitives as well as by providing a suitable set of abstract data types describing the most common data structures used in the parallel programming frameworks. The collective operations will be defined on the abstract data types, in a sense, although all of them will be independent of, or better, parametric in the data set managed/transformed by the collective operations. The set $C$ of control primitives used onto the data set at hand defines a graph of primitives that actually defines the whole application, too.

**Example 3.0.1** Let us suppose $C=\{\text{Apply}, \text{Fun}\}$, where $\text{Fun}$ is a primitive encapsulating a sequential function $f$ defined by the user, and $\text{Apply}$ is a primitive that applies an encapsulated primitive to each element of an input dataset. The graph of operators is represented by

$$G = \text{Apply}(\text{Fun}(f))$$

that applies the primitive $\text{Fun}(f)$ to all the elements of an input dataset. The graph of operators $G$ can be seen as depicted in fig.3.1. It concerns a tree rooted in the operator $\text{Apply}$. The root node has a number of children equal to the number of applications that the related operator performs. Thus, supposing that $n$ is the cardinality of the input data set $\{x_1, \ldots, x_n\}$, the root has $n$ children, each representing an application of the operator $\text{Fun}$. On the other hand, the operator $\text{Fun}$ is a primitive applying a function $f$ to its input. Thus, each $\text{Fun}$ node has a child representing the evaluation of such function.

In our proposal, each one of the collective operations is associated with a clear and easily composable formal semantics. By exploiting such semantics, the graph of primitives defining the whole parallel application can be easily understood and, possibly, modified by applying source-to-source semantic preserving transformation rules just to enhance or tune application parallel behavior and performance.

### 3.1 Abstraction mechanisms

A key role in our framework is played by iterators and views on abstract data types. In [GHJV95] we can find the definition of the iterator design pattern
The iterator provides an abstraction for accessing sequentially the elements of a collection of objects without exposing the inner structure of the collection itself.

In classical data driven approaches, iterators provide a powerful abstraction to organize and treat the access to data. Particularly, they allow to reason about data transformation in terms of collective operations, they can be ad hoc defined to solve particular access patterns, they can be concurrent, actually, rather that sequential, and, last but not least, their implementation is completely independent from the way we use them. The implementation only depends on the abstract data type representation.

Rather, in control driven approaches, iterators can be used as accessories of the control structures to define dependencies between components of the parallel application. Such dependencies are relative to the component computations rather than to the type of data processed or to the kind of access performed to such data.

For instance, let $M_1$ be a module (a function, a primitive, a process, a processor etc.) that depends from $M_2$, i.e. $M_1$ needs the values provided by $M_2$ in order to compute a result. A control driven approach focuses on the mechanisms by which $M_1$ and $M_2$ communicate the stream of values one to the other, leaving the responsibility to handle data concerns (for example, partitioning, gathering policies, joining etc.) to the programmer.

We argue that such concerns can be hidden through an iterator that carries out both the values and a semantics about how accessing them, and that becomes the actual value transmitted between $M_1$ and $M_2$.

Our idea is enforced by the following observation: iterators actually implicitly define or explicitly use a view of the abstract data type. Defining a view on a set of values, means giving them a logical organization related to their accessibility. For instance, an array view on the abstract data type collecting the values $\{x_1, \ldots, x_n\}$ means the possibility of indexing each element through a positional reference.

Iterators implicitly define a view, as the iterator defines an access pattern which is general with respect to the abstract data type definition and therefore forces on the abstract data type a sort of “access model”. For example, an iterator could return one-by-one all the elements of a collection as they were member of an array.

On the other hand, they use a view, in that a general access pattern defined by an iterator may be specialized in subtle ways to follow a view of the abstract data type. For example, a given array view may couple elements of the abstract data type as single members of its. In this case an iterator returning by definition the elements of the view one-by-one, will return couples of elements, just
because its behavior has been specialized by the particular access pattern related to the view of reference.

Let’s exemplify the concept through another example. Suppose to have an abstract data type implementing a generic set of values as well as a generic iterator enumerating all the elements of a given data structure. Suppose also that a matrix views exist, specialized to the given abstract data type. Such a view allows to define different type of access patterns, i.e. to declare which is the atomic unit of access, and to each type an iterator can be associated to get the units. Depending on the different application at hand, we can use the point access pattern (e.g. to compute the Mandelbrot set) or the row (column) access pattern (e.g. to compute matrix multiplication). In these cases, the iterator returns respectively an element of the matrix, a row or a column as a unit of data.

On the other side, in case we have two iterators, one accessing all the elements of a data collection sequentially and one accessing all them in parallel (concurrently), the usage of one of the two iterators define a view of the data structure at hand. In the former case, a set is viewed as a list while in the latter case it is viewed as a vector/array, in a sense.

Summarizing, we are delineating the basis of a model in which iterators can be used to encapsulate data access behaviors and such encapsulation can be exploited both in data-driven and in control-driven approaches.

3.2 Separation of concerns

Our further step will be the demonstration of how this power abstraction can be used to keep separated and independent data parallel and control parallel concerns while modeling and writing a parallel application.

**Data parallel concerns**  Abstract data types, views and iterators fully describe the data parallel concerns of the application. An 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 an abstract data type, 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. The former type is called parallel iterator while the latter is called sequential iterator and both types can be provided by a certain view.

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.

**Task parallel concerns**  The task parallel behavior of the application is described by using the set of primitives \( C \), 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 dependencies among modules. Such an application graph is constructed by selecting and composing the patterns provided by the set of primitives \( C \). \texttt{Pipe} (pipelines of stages), \texttt{Apply} (application of a function to all the elements of an input data set in parallel), \texttt{Seq} (sequential composition), \texttt{Fun} (sequential function) etc. populate \( C \).

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 an application can be written as a pipeline of two stages: the first stage \texttt{applies} function \( S_1(px) \) to all the pixels of the current
image; the second stage applies function $S_2$ 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 primitive computing a given sequential function.

**Orthogonalization** 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 way 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 couple 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 C by the Apply 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 (see chapter 4 and [Ske]). In our model, a farm computation can be described by combining the Apply 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.

### 3.3 Writing an application

Let us suppose we want to encode the graph given in fig.3.1.

The encoding of an application requires three steps, at least

1. The user has to define the data access concerns. Thus, he has to select/instantiate the appropriate abstract data types in order to represents its dataset of raw input values. He has also to select the views in order to define a logical structure for accessing them. For instance, he could decide to populate the input dataset $s$ with integer values $\{x_1, \ldots, x_n\}$ and to apply a matrix view on it of dimensions $r \times c$. This means that the values will be accessed by blocks of granularity $r$.

   A further step in defining data access concern is given by the definition of the iterators by which the given view can be accessed (and by which each control primitive will “bound” its input data). In case the elements of the array view are all independent, he could decide to access the array view through a parallel iterator $it_p$, thus exploiting the maximum degree of parallelism in accessing the view. Actually, together with the iterator type, the user has to select also the access pattern related to the given view. A pattern of access defines the dimension of each single access within the structure of the view or, it can define an order of access. For instance, on the array view we have patterns for accessing items by singletons or by blocks of a given granularity.

2. The user has to define the control concerns. Thus, it has to built its graphs of primitives by selecting some of them and composing/nesting them till the final application. In this case, mentioned in example 3.0.1, the graph $G$ is represented by an Apply encapsulating a Fun primitive. The latter type of primitives is responsible for the evaluation of a sequential function $f$ defined and provided by the user.
3. The user can proceed with the evaluation of the graph of primitives coupled with the iterator \( it_p \), e.g. it can execute its primitives in the order given by the dependencies defined in the graph. For simplicity we will call `evaluate` the function performing such evaluation.

Thus, our program will have the following structure:

```c
Program main(){
    /*------------- Data Parallel concerns/step 1 ----------------------*/
    // define the input data set
    s <- \{x1, x2, ..., xn\}
    // define an array view of s
    v <- ArrayView on s
    // define a parallel iterator with a given pattern of access
    itp <- ParIterator on v & AccessPattern on v

    /*------------- Control Parallel concerns/step 2 ---------------------*/
    // define the graph of primitives
    G <- Apply(Fun(f))

    /*------------- Coupling and Evaluation/step 3 ----------------------*/
    // execute the application graph
    evaluate(itp,G);
}
```

### 3.4 The semantics model

Still now, we have sketched a programming model in which data and task (or control) parallel concerns are kept orthogonal through a set of abstraction mechanisms.

A further goal of this thesis is also to provide a 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.

Moreover, we will exploit a way to establish if two application graphs are functionally equivalent or not. From our point of view:

**Definition 3.4.1** Two graphs of primitives are functionally equivalent if they denote both a function that, taking the same set of input arguments, returns the same set of output results.

The main contribution given by this work is a clear specification of the semantics and the introduction of a computational costs system provided by the semantic. In fact, the former formalizes both mechanisms for describing control concerns and data access concerns, thus reaching three goals:

1. 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

2. the semantics can assign some kind of evaluation costs about the evaluation of our abstract mechanisms in order to statically estimate the computational cost of the whole application. Particularly, each transformation \( E_1 \rightarrow E_2 \) assessing that graph expression \( E_1 \) “evolves” in the graph expression \( E_2 \), is labeled with a cost \( c \) related to such transformation. Thus, given a graph \( G \), we are able to assign a computational cost to \( G \) as a function of the computational costs of all the transformations involved.

3. such inference rules have also the role of rewriting rules between graph structures assessing if two graphs are functionally equivalent (see definition 3.4.1). Rewriting rules and static estimation of computational costs can drive static and dynamic improvements of the overall
performance. For example, our semantic framework prove 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 application graph at running or compile time. The choice of a graph of primitive \( G' \) alternative to \( G \), defined by the user, is established by comparing the computational costs of \( G \) and \( G' \). Instead, the structure of \( G' \) is lead by rewriting \( G \) through the system of inference rules.

Our programming model is fully described by the tuple

\[
M = \langle A, V, P, I, C \rangle
\]

where \( A \) stands for a set of abstract data types; \( V \) stands for the set of views on abstract data types; \( P \) stands for the set of access patterns defining unit of access; \( I \) stands for a set of iterators for accessing views; \( C \) stands for the set of control primitives which allow to describe structured graphs of operators (i.e. graphs built composing the available primitives).

Thinking about our abstract mechanisms in terms of a coordination model \( \langle E, L, M \rangle \), we can look at \( A, V \) and \( C \) as elements belonging to \( E \) just because they represents the entities to be coordinated; \( L \), the media by which coordination will be done is given by the iterator abstraction mechanism coupled with \( P \) abstractions. The semantic framework, here represented by the set of rules \( M \) to which the model ad hears, will be detailed in chapter 4 and 5 and it provides a formal, provable method to statically or dynamically evaluate the overall performance of the application and, if it is the case, to apply optimization rules.
Chapter 4

Accessing data

In this chapter we will detail the semantic representation of all the mechanisms abstracting on input data. In particular, we will give a formal description of abstract data types as sets of raw objects, views as structured set of objects, patterns of access and iterators as medium to access data.

Each abstract mechanism will be presented together with a set of operators to handle it. Clearly, such a set could be extended and modified but we are just intended to depict the main operations to handle data concerns, focusing on the semantic descriptions of data and accesses rather than on the implementation aspects of such descriptions.

First of all, we will give the formal definition of abstract data type as a flat set of values. If 

\[ s = \{s_1, \ldots, s_n\} \]

is a flat set of values, then \( s \in A \). A value of type \( A \) can be ground value, an objects whose type is defined by the user or a \( A \) object (thus, another flat set of \( A \) values). The following grammar describe all the possible values by which type \( A \) can be built (\( G \) stands for ground type and \( Object \) stands for a generic user-defined object type)

\[
G ::= \text{Integer} | \text{Float} | \text{String} | \text{Object} \\
A ::= G | \text{list of } A
\]

Table 4.1: A grammar defining the \( A \) type

On top of a given abstract data type a set of view can be instantiated. A view is an abstraction associating a logical structure to its abstract data type. We have to underline the term “logical” because, actually, the view is a kind of filter on the flat dataset representing an abstract data type. Such filter allows to refer the elements of the dataset with particular rules, without affecting its implementation.

For example, an ArrayView represents an homomorphism between elements is the dataset and elements in the view but the latter can be addressed by indexing their positions in the array. A split operation on an array view moves the element in position \( i \) to position \( i+1 \) without requiring the actual data movements. Moreover, if an abstract data type is built is order to be a flat set of values distributed across a network, a view on such abstract data type allow to refer the values as they were all on the same processor. Operations related to synchronization, caching etc. are matter of the implementation framework.

In this thesis we will consider ArrayView, MatrixView, TreeView, ListView and GraphView. For each of them, we will provide a formal definition and a set of operators for manipulating them and for instancing sequential and parallel iterators.

Iterators can be created on top of a given view by defining the access pattern, first. The access pattern is a structural declaration of the minimal unit of access, i.e. of the atomic block of data.
that the iterator will return. A sequential iterator will provide the such units declared on top of a
given view one-by-one. A parallel iterator will provide the same set as a whole.
Access patterns are defined as structured set of abstract data types

4.1 Abstract Data Type

As announced by the preamble of this chapter,

**Definition 4.1.1** An abstract data type is a flat set of $A$ values representing the input dataset. A
value of type $A$ can be a ground value, an object whose type is defined by the user or a list of $A$
values.

Given $n \in \mathbb{N}$, we formally define type $A$ as:

$$A \triangleq \{x_i \mid x_i \in A\}_{i \in [1,n]} = \{x_1, \ldots, x_n\}$$

By looking at the grammar in Tab. 4.1, it can be seen that $G$ refers to ground types.
A number of operators are defined to manipulate an abstract data type object and they are defined
in Tab. 4.2. Since $A$ data set is a flat set of values, the most operators represent canonical
operations working on sets. For instance, `append` appends a new object on the current abstract
data type\(^{1}\).

**Example 4.1.1** A raw input dataset could be represented by a collection of `int` values

$$s = \{1, 2, \ldots, 100\}$$

or by a collection of complex objects. For instance, let `ComplexNumber` be an object type defined by the
user as a pair $(x, y)$ where $x, y \in \mathbb{R}$ represent, in component notation, the complex value
$z = x + iy$. Then, a raw input data set of `ComplexNumber` objects is given by the set

$$s = \{(x, y) \mid (x, y) \in \text{ComplexNumber}\}$$

**Example 4.1.2** A raw input dataset could be represented by a set of abstract data values such as, for
instance,

$$s' = \{\{1, 2, 3\}, \{4, 5, 6\}, \ldots, \{98, 99, 100\}\}$$

where $\{1, 2, 3\}, \{4, 5, 6\}, \ldots, \{98, 99, 100\}$ are separately $A$ values.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$s'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$A$</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$A \times \rightarrow$</td>
<td>$A \times \rightarrow$</td>
</tr>
<tr>
<td>$N \rightarrow$</td>
<td>$N \rightarrow$</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$A \times \rightarrow$</td>
<td>$A \times \rightarrow$</td>
</tr>
<tr>
<td>$N \rightarrow$</td>
<td>$N \rightarrow$</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$A \times \rightarrow$</td>
<td>$A \times \rightarrow$</td>
</tr>
<tr>
<td>$N \rightarrow$</td>
<td>$N \rightarrow$</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$A \times \rightarrow$</td>
<td>$A \times \rightarrow$</td>
</tr>
<tr>
<td>$N \rightarrow$</td>
<td>$N \rightarrow$</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

Table 4.2: Operators on ADT's

---

\(^{1}\) Another symbol for such operator that will be used in the sequel of this thesis is $\cup$, i.e.
append$(\{s_1, \ldots, s_2\}, \{t_1, \ldots, t_2\}) = \{s_1, \ldots, s_2\} \cup \{t_1, \ldots, t_2\} = \{s_1, \ldots, s_2, t_1, \ldots, t_2\}$.
4.2 Views

Definition 4.2.1 A view (on an abstract data type) is a structural and logical organization of the abstract data type content

The adjective “structural” refers to the fact that a view provides one or more structural relations between the abstract data type elements. For instance, a view could provide access methods by means of a father-child relation. The adjective “logical” means that there is no an immediate relation between the structure given by the view and the actual implementation of the input dataset itself. The structuring is just a “logical” commodity for exploiting particular operators provided by and related to the view at hand.

In this thesis we will consider the following set of views:

\[ \mathcal{V} = \{ \text{ArrayView}, \text{MatrixView}, \text{TreeView}, \text{ListView}, \text{GraphView} \} \]

each of which is going to be analyzed in the following sections.

4.2.1 ArrayView

An ArrayView type on an abstract data type allows the user/programmer to access data items in the dataset as if the dataset was an array data structure, thus explicitly indexing their contiguous position in the view. Formally, the ArrayView type on \( s \in \mathcal{A} \) is defined as an homomorphism between elements of \( s \) and elements of the view, thus:

\[
\text{ArrayView}(s) \triangleq \{ a_i \mid \forall i \in [1, \text{size}(s)], a_i = s_i \}
\]

In Table 4.3 are listed the operators provided on this view type and, of course, they reflect the well known operator on the array data structure (set(i, v) and get(i)), and the operator for getting parallel and sequential iterators. In the table, the notation \( v[\frac{i}{j}] \) has been used to indicate an array view \( v \) whose \( i \)-th position has been updated in order to contain the object \( o \). Moreover, an iterator object is represented as a pair \((\sigma, 0)\) whose semantic will be explained in section 4.4.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>( \mathcal{V} \times \mathbb{N} \to \mathcal{A} )</td>
<td>( \mathcal{A} )</td>
</tr>
<tr>
<td>set</td>
<td>( \mathcal{V} \times \mathcal{A} \times \mathbb{N} \to \mathcal{V} )</td>
<td>( \mathcal{V} )</td>
</tr>
<tr>
<td>getParIterator</td>
<td>( \mathcal{V} \times \text{AccessPattern} \to \mathcal{I} )</td>
<td>( \mathcal{I} )</td>
</tr>
<tr>
<td>getSeqIterator</td>
<td>( \mathcal{V} \times \text{AccessPattern} \to \mathcal{I} )</td>
<td>( \mathcal{I} )</td>
</tr>
</tbody>
</table>

Table 4.3: Operators on ArrayView

Example 4.2.1 Let \( s \) be the abstract data type defined in the example 4.1.1. An ArrayView on \( s \) is represented as:

\[
\text{ArrayView}(s) = < 1; 2; 3; 4; \ldots; 100 >
\]

4.2.2 MatrixView

A view of type MatrixView allows the user/programmer to access data items in the flat dataset of length \( n \) as if the dataset was a matrix. This means that, if the matrix has dimension \( r \times c \) where \( n = r \times c \), the elements in the dataset will be accessed by blocks of \( r \) elements in order to provide a row of the matrix (in all those cases in which \( n = r \times c + q \) and \( q \neq 0 \) we could assume one more row of \( q \) elements \(^2\)).

Given \( r, c \in \mathbb{N}^+ \), the MatrixView type is formally defined as:

\[
\text{MatrixView}(s, r, c) = \{ \{ t_0, \ldots, t_{r-1} \} \mid \forall i \in [0, r - 1], t_i = \{ s_{c+1+j} \}_{j \in [1, c]} \}
\]

\(^2\)In order to avoid heaviness in the notation, we will always assume that \( n = r \times c \), also because these type of details will be solved at implementation level.
Tab. 4.4 summarizes the operator related to this type of view. As it can be expected, operators for factoring parallel and sequential iterators are provided. Such operators will be more clear in the next sections, as well as the explanation of the pair representing the iterator type.

**Example 4.2.2** Let $s$ be the *abstract data type* defined in the example 4.1.1. A $10 \times 10$-matrix could be defined by setting up

$$M_1 = \text{MatrixView}(10, 10) = \langle \langle 1; 2; \ldots; 10 >_{>}; \ldots; < 91; 92; \ldots; 100 >_{>}\rangle$$
$$M_2 = \text{MatrixView}(2, 50) = \langle \langle 1; 2; \ldots; 50 >_{>}; \langle 51; 52; \ldots; 100 >_{>}\rangle$$

| cols : MatrixView $\rightarrow$ N | cols($\text{MatrixView}(r, c)$) = $c$ |
| rows : MatrixView $\rightarrow$ N | rows($\text{MatrixView}(r, c)$) = $r$ |
| getParIterator : $\mathcal{V} \times \text{AccessPattern} \rightarrow I$ | getParIterator($v$) = ($\sigma, 0$) |
| getSeqIterator : $\mathcal{V} \times \text{AccessPattern} \rightarrow I$ | getSeqIterator($v$) = ($\sigma, 0$) |

**Table 4.4: Operators on MatrixView**

### 4.2.3 List View

A view of type *List View* logically represents a linear abstract data structure built by a sequence of elements $l = [l_1; \ldots; l_n]$. As in the case of the *Array View* view, building a *List View* on an abstract data set $\{s_1, \ldots, s_n\}$ means applying an isomorphism from $\mathcal{A}$ to *List View*. In fact, the *List View* type is defined as:

$$\text{ListView}(s) \triangleq \{l_i \mid \forall i \in [0, \text{size}(s)][l_i = s_i]\}$$

But a list view differs from an array view with respect to the available operators to access and modify it. In Tab. 4.5 the operations on the *List View* type are listed. As it can be expected, they reflect the canonical operators on lists ($\text{hd}$, $\text{tl}$ and $\text{lenght}$ returning the head, the tail and the length of a list, respectively) and the operators for factoring sequential and parallel iterators whose type will be more clear in the next sections.

| hd : ListView $\rightarrow$ $\mathcal{A}$ | hd($[l_1; \ldots; l_n]$) = $l_1$ |
| tl : ListView $\rightarrow$ $\mathcal{A}$ | tl($[l_1; \ldots; l_n]$) = $l_n$ |
| lenght : $\mathcal{V} \rightarrow$ N | lenght($[l_1; \ldots; l_n]$) = $n$ |
| getParIterator : $\mathcal{V} \times \text{AccessPattern} \rightarrow I$ | getParIterator($v, a$) = ($\sigma, 0$) |
| getSeqIterator : $\mathcal{V} \times \text{AccessPattern} \rightarrow I$ | getSeqIterator($v, a$) = ($\sigma, 0$) |

**Table 4.5: Operators on List View**

**Example 4.2.3** Let $s$ be the *abstract data type* defined in the example 4.1.1. An isomorphic list on $s$ is trivially given by the list $L$ built as follows:

$$L = [1; 2; 3 \ldots; 100]$$

### 4.2.4 Graph View

A graph is an abstract data type logically represented by a pair $(\mathcal{V}, \mathcal{E})$. $\mathcal{V}$ provides the set of vertexes, $\mathcal{E}$ represents a set of edges between pairs of vertexes. The application of a *Graph View* view on an input dataset $s$ generates a *Graph View* object in which the set of vertexes is isomorphic with the dataset and a starting set of nodes is provided by the user. Such object can be successively manipulated through the operators on the graph structure provided by the framework. The *Graph View* type is defined as follows:
4.2. VIEWS

GraphView ≜ Vertex list × Edge list

where the symbol “×” is used as constructor for pairs of objects. Moreover,

Vertex ≜ A and Edge ≜ Vertex × Vertex

Vertex and Edge are respectively the constructors for vertexes and edges. Furthermore, for each element of the input dataset, an homomorphism between \( v \in \text{Vertex} \) and \( s_i \in A \) represents the vertex “viewing” the \( i \)-th element of the input data set. On the other hand, a pair \((v_1, v_2)\) for some \( v_1, v_2 \in \text{Vertex} \) represents an edge linking vertex \( v_1 \) and vertex \( v_2 \).

Such a characterization defines an undirected and vertex-labeled graph\(^3\). In order to insert a unique direction to edges and/or labels to edges, we simply have to enrich the type \( \text{Edge} \). For instance, an edge-labeled graph (i.e. a weighted graph) can be described by adding a weight \( w \) to the pair defining an edge. The direction is given by introducing the constraint \((v_i, v_j) \neq (v_j, v_i)\).

Tab. 4.6 provides a summary of the main operators on graph, including the constructors for the iterator type \( I \).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>GraphView → Bool</td>
</tr>
<tr>
<td>insVtx</td>
<td>GraphView × Vertex → GraphView</td>
</tr>
<tr>
<td>insEdge</td>
<td>GraphView × Edge → GraphView</td>
</tr>
<tr>
<td>getParVtxIterator</td>
<td>GraphView × AccessPattern → I</td>
</tr>
<tr>
<td>getSeqVtxIterator</td>
<td>GraphView × AccessPattern → I</td>
</tr>
<tr>
<td>getParEdgeIterator</td>
<td>GraphView × AccessPattern → I</td>
</tr>
<tr>
<td>getSeqEdgeIterator</td>
<td>GraphView × AccessPattern → I</td>
</tr>
</tbody>
</table>

**Example 4.2.4** As a first construction, the application of a GraphView view creates a graph structure modifiable by the well known operations on graphs summarized in Tab. 4.6 and built starting from an initial set of edges the programmer provides.

Let \( S = \{a, b, c, d\} \) the input data set. A possible configuration of a GraphView is given by

\[
\text{GraphView}(s, \{(a, b); (b, c); (c, d)\}) = (V, E) = ([a, b, c, d], [(a, b); (b, c); (c, d)])
\]

In this case the graph is a chain of vertexes in which each vertex (but the first and the last) is linked just with its predecessor and its successor (literally speaking).

4.2.5 TreeView

A tree is an undirected, connected, acyclic graph. As a consequence, the definition of a TreeView is very similar to the one given for the GraphView.

The TreeView type is defined as a pair whose first element is the set of vertexes called nodes and second element is the set of edges between pairs of nodes:

TreeView = (N, E) = Node list × Edge list

Also in case of a TreeView, the type Node represents the mapping between an element of the input abstract data type and a node of the view, while the Edge type represents a link between pairs of nodes.

\(^3\)A graph can be also a edge-labeled graph if the edges are labeled
Because of its structure, such a view could provide a very high number of access patterns and they can be associated both with parallel and with sequential iterators. For instance, a pattern of type LevelOrderPattern (see Tab.4.7) establishes that the nodes representing the tree will be accessed level by level. A PreOrderPattern defines a pre-order listing of the tree nodes. Iterators associated with these type of patterns can provide the nodes sequentially or in parallel and coherently with the order given by the defined access pattern.

<table>
<thead>
<tr>
<th>isEmpy : TreeView → Bool</th>
<th>isEmpty(t) = (size(fst(t)) = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insNode : TreeView × Node → TreeView</td>
<td>insNode(t, v) = (append(fst(t), v), snd(t))</td>
</tr>
<tr>
<td>insEdge : TreeView × Edge → TreeView</td>
<td>insEdge(t, e) = (fst(t), append(snd(t), e))</td>
</tr>
<tr>
<td>getParNodeIterator : TreeView × AccessPattern → I</td>
<td>getParNodeIterator(t) = (σ, 0)</td>
</tr>
<tr>
<td>getSeqNodeIterator : TreeView × AccessPattern → I</td>
<td>getSeqNodeIterator(t) = (σ, 0)</td>
</tr>
<tr>
<td>getParEdgeIterator : TreeView × AccessPattern → I</td>
<td>getParEdgeIterator(t) = (σ, 0)</td>
</tr>
<tr>
<td>getSeqEdgeIterator : TreeView × AccessPattern → I</td>
<td>getSeqEdgeIterator(t) = (σ, 0)</td>
</tr>
</tbody>
</table>

Table 4.7: Operators on TreeView

4.3 Characterizing the access through patterns of access

As mentioned before, a view provides a logical structure to the raw data set of input values. It is also worthwhile that each view should provide different patterns for accessing to the values it represents. As an example, after having defined a MatrixView of size $n \times m$, the application should require to access the elements of the matrix by rows, by columns or by sub-blocks statically or dynamically sized.

Analogously, a TreeView should be accessed by level or by deep; the ArrayView’s elements could be accessed coupling them etc.

As shown by these trivial examples, a mechanism to define the access unit of a given view is needed. In order to allow different way for accessing the inner elements, each view provides a set of patterns allowing to read/write pre-defined basic blocks of data. Further, each view encapsulates a default pattern of access that is the one providing the elements of the view one by one as a sequential iteration would allow. For example, the default access pattern of a $(n \times m)$-MatrixView allows to access $n \times m$ elements addressed through their related row and column indexes.

Tab.4.8 summarizes the access patterns provided by our views. Also some of them are described in details by the following sections, since they corresponds to the state of the actual implementation framework (that will be presented in chapter 9.

4.3.1 Array Pattern

**Definition**  Let $v \in \text{ArrayView}$ and $g \in \mathbb{N}$ the grain of the $i$-th access unit. Then, the array pattern allows the user/programmer to access the items of the array view by blocks of grain $g$. Formally,

$$\text{ArrayPattern}(v, g) = \{a_i \mid \forall i \in [0, \frac{\text{size}(v)}{g} - 1].a_i = v_i \times g + 1; \ldots; v_i \times g + g\}$$

Again, we assume $\text{size}(v) = g \times c$ for some $c \in \mathbb{N}^+$, providing that the odd cases are handled at implementation time.
4.3. CHARACTERIZING THE ACCESS THROUGH PATTERNS OF ACCESS

<table>
<thead>
<tr>
<th>View</th>
<th>AccessPattern</th>
<th>Unit of access</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayView</td>
<td>ArrayPattern</td>
<td>a block of array elements</td>
</tr>
<tr>
<td>MatrixView</td>
<td>PointPattern</td>
<td>a square block of matrix elements</td>
</tr>
<tr>
<td></td>
<td>RowPattern</td>
<td>a block of rows</td>
</tr>
<tr>
<td></td>
<td>ColPattern</td>
<td>a block of columns</td>
</tr>
<tr>
<td>ListView</td>
<td>ListPattern</td>
<td>a set of list elements</td>
</tr>
<tr>
<td>GraphView</td>
<td>VertexPattern</td>
<td>a vertex of the graph</td>
</tr>
<tr>
<td></td>
<td>EdgePattern</td>
<td>an edge of the graph</td>
</tr>
<tr>
<td></td>
<td>NeighborPattern</td>
<td>a vertex and the set of its neighbors.</td>
</tr>
<tr>
<td>TreeView</td>
<td>NodePattern</td>
<td>a node of the tree</td>
</tr>
<tr>
<td></td>
<td>EdgePattern</td>
<td>an edge of the tree</td>
</tr>
<tr>
<td></td>
<td>ChildrenPattern</td>
<td>a parent node and its children</td>
</tr>
<tr>
<td></td>
<td>PreOrderPattern</td>
<td>a node of the tree</td>
</tr>
<tr>
<td></td>
<td>InOrderPattern</td>
<td>a node of the tree</td>
</tr>
<tr>
<td></td>
<td>PostOrderPattern</td>
<td>a node of the tree</td>
</tr>
<tr>
<td></td>
<td>LevelOrderPattern</td>
<td>a level of the tree</td>
</tr>
</tbody>
</table>

Table 4.8: Pattern of access on the set of views

**Description**  This pattern allows to access the element of the input data set by referring their relative position into the dataset. Each pattern can be defined providing a grain as access dimension, meaning that each element read/written by the access pattern is built by grouping adjacent elements of the original view.

**Provided operators**  The provided operations are listed in Tab.4.9 where it appears the notation $ap[i]$ indicating that the position $i$ of $ap$ is overwritten by the value $o$.

<table>
<thead>
<tr>
<th>get</th>
<th>ArrayPattern $\times \mathbb{N}$ $\rightarrow \mathcal{A}$</th>
<th>get($ap, i$) = $a_p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>ArrayPattern $\times \mathcal{A} \times \mathbb{N}$ $\rightarrow$ ArrayPattern</td>
<td>set($ap, o, i$) = $ap[i^o]$</td>
</tr>
<tr>
<td>length</td>
<td>ArrayPattern $\times \mathbb{N}$</td>
<td>length([$ap_0; \ldots; ap_{n-1}$]) = $n$</td>
</tr>
</tbody>
</table>

Table 4.9: Array Pattern

**Example 4.3.1**  Let $v \in$ ArrayView be the array view defined in the example 4.2.1. The definition of an array pattern of grain 2 leads to a collection of pairs of $v$’s elements as the following:

$$\text{ArrayPattern}(v, 2) = \overline{\nu} = \{(1; 2), (3; 4), \ldots, (99; 100)\}$$

so that

$$\text{get}(\overline{\nu}, 4) = [9; 10]$$

**4.3.2 Pattern for accessing matrices - ByPointPattern**

**Definition**  The ByPointPattern pattern for matrices defines the “grain” by which accessing the elements of a given matrix. Such grain is given addressing sub-matrices, thus declaring a pair of values $(r \in \mathbb{N}^+, c \in \mathbb{N}^+)$, the former indicating the number of rows, the latter indicating the number of columns of the $i$-th element of the pattern, thus of the sub-matrix dimensions.

**Description**  Let $v \in$ MatrixView, a matrix view. Let $rg, cg \in \mathbb{N}$ be respectively the number of rows and the number of columns of the $i$-th block. Let $nr = \frac{\text{rows}(v)}{rg}$ and $nc = \frac{\text{cols}(v)}{cg}$ be respectively
the number of sub-blocks on the $v$’s rows and on the $v$’s columns. Then, the ByPointPattern’s collection is defined as:

$$\text{ByPointPattern}(v, r_g, c_g) = \{ t_{i,j} \mid \forall i \in [0, nr - 1], \forall j \in [0, nc - 1], t_{i,j} = M(i, j) \}$$

where

$$M(i, j) = \{ v(l_1, l_2) \mid \forall x \in [0, r_g - 1], l_1 = i \times r_g + x \land (\forall y \in [0, c_g - 1], l_2 = j \times c_g + y) \}$$

Provided operators

The operators are listed in Tab. 4.10.

| get : $\text{ByPointPattern} \times \mathbb{N} \times \mathbb{N} \to \mathbb{A}$ | get$(a, i, j) = a_{(i,j)}$ |
| set : $\text{ByPointPattern} \times \mathcal{O} \times \mathbb{N} \times \mathbb{N} \to \text{ByPointPattern}$ | set$(a, o, i, j) = a_{(i,j)}$ |
| size : $\text{ByPointPattern} \to \mathbb{N}$ | size$(\{a_0, \ldots, a_{n-1}\}) = n$ |

Table 4.10: ByPointPattern operators

Example 4.3.2

Let $M \in \text{MatrixView}$ be a $(4 \times 6)$-matrix view defined on an input dataset of int values.

$$M = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
7 & 8 & 9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 & 17 & 18 \\
19 & 20 & 21 & 22 & 23 & 24
\end{pmatrix}$$

The ByPointPattern pattern allows to access sub-matrices of $M$. For example, we can instantiate a $\text{ByPointPattern}(M, 2, 2)$ that creates a $2 \times 3$ grid of six sub-matrices:

$$\text{ByPointPattern}(M, 2, 2) = \overline{M} = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
7 & 8 & 9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 & 17 & 18 \\
19 & 20 & 21 & 22 & 23 & 24
\end{pmatrix}$$

each of which can be accessed indexing its relative position into the grid, as for instance,

$$\text{get}(\overline{M}, 1, 1) = \overline{M}_{(1,1)} = \{ v_{(2,2)}, v_{(2,3)}, v_{(3,2)}, v_{(3,3)} \} = \{15; 16; 21; 22\}$$

It’s worthwhile that $\overline{M}_{(1,1)} \in \mathbb{A}$, thus a new, different view can be applied on it in order to refine the access mode.

4.3.3 Pattern for accessing matrices - ByRowPattern

Definition

The ByRowPattern pattern allows to access the original $(n \times m)$-matrix view by rows. Each block the pattern accesses can be built from one to N rows of the original matrix. Such “grain”, i.e. the number of rows of each block, is provided within the pattern constructor. Hence, given $v \in \text{MatrixView}$ and let $r_g \in \mathbb{N}$ be the grain value, the definition of the ByRowPattern is given as follows:

Description

$$\text{ByRowPattern}(v, r_g) = \{ t_i \mid \forall i \in [0, \frac{\text{rows}(v)}{r_g} - 1] \}$$

where

$$t_i = \{ v(x,y) \mid \forall h \in [0, r_g - 1], x = r_g \times i + h \land (\forall k \in [0, \text{cols}(v) - 1], y = k) \}$$
4.3. CHARACTERIZING THE ACCESS THROUGH PATTERNS OF ACCESS

Table 4.11: ByRowPattern operators

<table>
<thead>
<tr>
<th>Provided operators</th>
<th>The provided operators are listed in Tab. 4.11</th>
</tr>
</thead>
</table>

Example 4.3.3 Let $M$ be the $(4 \times 6)$-matrix view defined in Example 4.3.2. The pattern $\text{ByRowPattern}(M, 1)$ allows to access the matrix by single row:

$$\text{ByRowPattern}(M, 1) = \overline{M}_1 = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
7 & 8 & 9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 & 17 & 18 \\
19 & 20 & 21 & 22 & 23 & 24
\end{pmatrix}$$

so that

$$\text{get}(\overline{M}_1, 2) = [13; 14; 15; \ldots; 18];$$

In the same way, $\overline{M}_2 = \text{ByRowPattern}(M, 2)$ sets blocks of two rows as access unit, then:

$$\text{get}(M_2, 2) = [13; \ldots; 18; 19; \ldots; 24]$$

4.3.4 Pattern for accessing matrices - ByColPattern

Definition The ByColPattern pattern allows to access the original $(n \times m)$-matrix view by columns. Each block the pattern accesses can be built from one to $M$ columns of the original matrix. Such “grain”, i.e. the number of columns of each block, is provided within the pattern constructor. Hence, given $v \in \text{MatrixView}$ and let $cg \in \mathbb{N}$ be the grain value, the definition of the ByColPattern is given as follows:

Description

$$\text{ByColPattern}(v, cg) = \{v_{j, \ast} \mid \forall j \in [1, \frac{\text{cols}(v)}{cg} - 1]\}$$

where

$$v_{j, \ast} = \{v(x,y) \mid \forall l_1 \in [0, \text{rows}(v) - 1], x = l_1 \land (\forall h \in [0, cg - 1], y = cg \times j + h)\}$$

Provided operators The provided operators are listed in Tab. 4.12.

Table 4.12: ByColPattern operators

<table>
<thead>
<tr>
<th>Provided operators</th>
<th>The provided operators are listed in Tab. 4.12.</th>
</tr>
</thead>
</table>

Example 4.3.4 Let $M$ be the $4 \times 6$-matrix view defined in the Example 4.2.2. The pattern $\text{ByColPattern}(M, 2)$ allows to access the matrix by single column:

$$\text{ByColPattern}(M, 1) = \overline{M}_1 = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
7 & 8 & 9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 & 17 & 18 \\
19 & 20 & 21 & 22 & 23 & 24
\end{pmatrix}$$

so that

$$\text{get}(\overline{M}_1, 2) = [3; 9; 15; 21];$$

In the same way, $\overline{M}_2 = \text{ByColPattern}(M, 2)$ sets blocks of two columns as a single access unit, then:

$$\text{get}(M_2, 2) = [3; 4; 9; 10; 15; 16; 21; 22];$$
4.4 Iterators

The constructor for the *iterator* type is given as follows:

\[ I : \text{AccessPattern} \rightarrow \text{AccessPattern} \times \text{Pointer} \]

The iterator type is a pair \((\sigma, p)\) where \(\sigma \in \text{AccessPattern}\) and \(p \in \text{Pointer}\). The value \(p\) “points” to the last element accessed through the iterator’s primitives. The nature of the access unit pointed by the iterator depends exclusively from the logic embedded by the access pattern \(\sigma\) as well as the implementation of the type \(\text{Pointer}\).

**Example 4.4.1** Let us consider the example 4.3.1 in which an array view \(v\) of 100 elements is given and let \(ap = \text{ArrayPattern}(v, 2)\) be the defined access pattern. Pattern \(ap\) sets the access units of the array to be disjoint pairs of adjacent elements. An iterator \(it\) that allows to access such pairs as a singleton is formally represented by the pair \((ap, p)\) such that

\[
\text{it} = (ap, p) = (\text{ArrayPattern}(v, 2), p) = (\{[1; 2], [3; 4], \ldots, [99; 100]\}, 1)
\]

In this case, since the array structure can be easily indexed with natural numbers, the \(\text{Pointer}\) type can be represented by numerical values.

As the iterator is the key abstraction in our orthogonalization process, the operators associated with iterators are crucial. The basic idea while using an iterator is the one coming from the design pattern tradition: iteratively a boolean operator, namely \(\text{hasNext()}\), is queried for the availability of new elements into the collection: if there are elements to be accessed, one of them is returned by the appropriate operator (i.e. \(\text{current()}\)) and the pointer moves to the next element (i.e. \(\text{skip()}\)); else, no element are returned and the iteration process stops. The usage of a (sequential) iterator can be sketched as follows:

```plaintext
while (hasNext(it)) // ask for a new element in the collection
{
    d = current(it); // retrieve the current element
    ...
    skip(it); // skip the pointer to the next element (if any)
}
```

The operators on the iterator type supported by our formal framework are summarized in Tab. 4.13 and they can be described as follows:

- **current**: \(I \rightarrow A\) is the operator that returns the current element (if any) of the collection pointed by the iterator provided as input.
- **hasNext**: \(I \rightarrow \text{bool}\) returns true if the inner pointer points to a valid element of the collection (that is, there is at least one element \(\text{current}\) can return if invoked)
- **setCur**: \(I \times A \rightarrow I\) sets in the position pointed by the iterator the new abstract data type given as input. The results is the iterator pointing to the updated collection
- **size**: \(I \rightarrow \mathbb{N}\) return the number of elements belonging to the collection
- **skip**: \(I \rightarrow I\) returns an iterator pointing to a successive position of the collection (if any) respect to the iterator given as input

In our formal framework the semantics and the usage of such a powerful abstract mechanism has been kept the same as it is in more standard and well-known environments, a part from a further distinction we have introduced aiming at classifying iterators between *parallel* and *sequential* ones.
### 4.5 Related work

The whole framework design is based on the concept of iterators, views and collective operations that all represent, in different frameworks, assessing concepts. By merging them into our framework we simply extended the possibilities they offer when kept alone. Iterators and views are common concepts in the OO world. C++ standard template library provides iterators, Java.util

---

**Table 4.13: Operators for the I type**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>curr : I → A ∪ O</td>
<td>curr((ap, p)) = get(ap, p)</td>
</tr>
<tr>
<td>hasNext : I → bool</td>
<td>hasNext((ap, p)) = (size(ap) = p)</td>
</tr>
<tr>
<td>set_curr : I × O → I</td>
<td>set_curr((ap, p), o) = set(ap, o, p)</td>
</tr>
<tr>
<td>size : I → N</td>
<td>size((ap, p)) = size(ap)</td>
</tr>
<tr>
<td>skip : I → I</td>
<td>skip((ap, p)) = (ap, p + 1)</td>
</tr>
</tbody>
</table>

**Parallel vs Sequential iterators**  As we have mentioned before, the iterator is the key abstract mechanism that allow us to encapsulate data parallel concerns. In a sense, a view is a declaration about what we want logically to access; an access pattern defines in which order we want to access; since we are talking about parallel computation, we are missing a definition of how we want to access, that is to access in parallel or sequentially. The distinction between parallel and sequential iterators fills the gap.

The idea is that once a given view and the related unit of access have been fixed, we have just to declare if retrieving the elements one-by-one (i.e. sequentially) or as a whole (i.e. in parallel).

Such a declaration is given by specializing the type of the iterator. As a consequence, the current operator invoked on a sequential iterator will return the elements of the collection one-by-one. The current operator invoked on a parallel iterator will return all the element of the collection in a whole, as they were accessed all together. Formally, the given it = (ap, p) ∈ I, the current operator is defined as follows:

\[
\text{current}(\text{it}) = \begin{cases} 
\{\text{get}(\text{ap}, i)\}_{i \in [0,\text{size}(\text{ap})]} & \text{if } \text{hasNext}(\text{it}) \text{ and } \text{it} \in \text{ParIterator} \\
\text{get}(\text{ap}, p) & \text{if } \text{hasNext}(\text{it}) \text{ and } \text{it} \in \text{SeqIterator} \\
\bot & \text{if not } \text{hasNext}(\text{it}) 
\end{cases}
\]

where the symbol \(\bot\) denotes an access error related to the absence of available elements to return.

**Example 4.4.2**  Let us take into account the access pattern ByPointPattern(M,2,3) defined in the example 4.3.2. A sequential iterator it_s defined on top of such an access pattern allows to retrieve bi-dimensional blocks of the input matrix one-by-one. In fact, let current\(^{(i)}\)(it_s) be the \(i\)-th call to the operator current (or, analogously, let \(i\) be the position pointed by the current state of the iterator it_s); after six sequential calls all the element of the view have been accessed.

\[
\begin{align*}
\text{current}^{(1)}(\text{it}_s) &= \{1, 2, 7, 8\} \\
\text{current}^{(2)}(\text{it}_s) &= \{3, 4, 9, 10\} \\
\vdots \\
\text{current}^{(6)}(\text{it}_s) &= \{17, 18, 23, 24\}
\end{align*}
\]

Instead, on the basis of the same access pattern, a parallel iterator it_p could be defined. In this case, all the access units of the matrix are retrieved once a time, thus

\[
\text{current}^{(1)}(\text{it}_p) = \{\{1, 2, 7, 8\}, \{3, 4, 9, 10\}, \ldots, \{17, 18, 23, 24\}\}
\]

while for each \(i > 1\), current\(^{(i)}\) = \(\bot\).

Thus, in a sense, the type of the iterator introduce a run time behavior of the data access that will be completed sequentially or in parallel for each element provided by the access pattern.
collection data types provide iterators, some STL implementations offer the view concept. Collective operations, on the other side, play a central role in the data parallel programming model world as well as in the whole skeleton framework.

In the field of parallel programming there are many examples of clever iterator usage. Several works are extension of the Standard Template Library as for example [JGB97, RHC+96] and their design is heavily influenced by its predecessor. All of them offer some parallel implementation of iterators, containers and generic parallel algorithms. NESL[BH99] and some following works has reached a good degree of nested parallelism capability but the user intervention is still heavily needed.

STAPL (Standard Template Parallel Library) [AJR+01, RAO98] inherits most of the Standard Template Library’s principles. Programmers write parallel algorithms called pAlgorithms whose inputs are pRanges (sort of parallel iterators); pRanges are provided by pContainers (parallel STL containers), which guarantee access control to their encapsulated data through pRanges and/or conventional STL iterators. STAPL is very near to our idea of using iterator as access control mechanism, in particular for data parallel application. Nevertheless, it completely misses control flow features to exploit control/task parallelism.

Kuchen’s skeleton library [Kuc02] provides type abstractions for the distribution and the partitioning of data structures; after the data has been statically distributed, on the basis of the selected distribution policy, the library provides operations to access it as a whole or to access a particular partition, regardless all communication details. This approach implies a sort of conceptual awareness about the distribution, explicit communication calls to manage overlapped partitions and the possibility to program parallel programs is strictly related to the availability of the data structures and policies provided by the library.
Chapter 5

The semantic framework: control parallelism

Primitives of control are the abstraction mechanism by which describing control concerns of a parallel application into our framework. In this chapter we are going to describe a minimal, extensible set of primitives each of which has a precise semantics and behavior. These primitives can be composed and nested in order to design an application graph whose parallel behavior results from the composition of its components’ behavior.

The semantics of each primitive is provided through of a description of its functional behavior and a description of its evaluation process. Two functions are introduced representing respectively the evaluation of the graph and the evaluation of operators on iterators inside the framework.

In the next chapter we will provide the evaluation semantics for each collective in terms of inference rules, demonstrating how much this notation is the most appropriate in describing the evaluation process step by step and hence, transformation per transformation.

5.1 Control primitives

Control parallelism is described by a set of control primitives representing structured patterns of control. Each primitive accepts an iterators as input parameter. The final application is given by simply selecting and combining/nesting primitives, thus instantiating a control graph. Thus, the parallel behavior of such a graph depends on the primitives composing the graph itself as well as on the iterators encapsulating data accesses.

The key of orthogonality  The basic idea we are meant to catch is that each node composing an application graph “is responsible for” (or “views”) only a certain portion of the input data space (that is the one flowing along its own path). Such space should be addressed and accessed by each node independently with respect to all the other nodes. This is just the hardest step in making data concerns orthogonal with respect to control concerns since, at this point, the latter should be treated at the same level of the former ones.

As an example, let us split a matrix of pixels into blocks of rows and then, let us split each row into singletons, in order to apply the Mandelbrot algorithm to each pixel. In this case, each processing element evaluating a single row has only the “view” of a portion of the input data space (its own row). If we don’t provide abstraction mechanisms on the row representation, each processing element have to split such a row into pieces, gather them to all the processing elements collaborate in the application and, finally, collect them in order to provide the final (partial) result. In this case data concerns (splitting and joining piece of data) and control concerns (gathering and collecting messages for exchanging data), are quite dependent because each of them depends on how the other one has been written and organized.
Indeed, as mentioned before, in our framework each selected primitive accepts one iterator as input parameter. These iterators completely encapsulate the data access behavior that will be embedded by the primitive receiving it as functional argument. As a consequence, the same control graph can be applied to different iterators, i.e. we can evaluate the same parallel function accessing data in a different manner. Conversely, we can organize the access to our data in a stable, efficient manner and then applying on them different control graphs in order to enlarge the spectrum of parallel solutions.

Such a flexibility is the key for making concrete the plain orthogonality between data access and control directives. Moreover, coupling a certain primitive with a sequential w.r.t. a parallel iterator can lead to a different exploitation of parallelism, as it will be seen in the following sections.

It is worthwhile that the set of primitives presented in this chapter is not intended to be complete and exhaustive. We have studied just the basic primitives known by a parallel programming framework in order to prove the feasibility of our theory and to start working with quite general, well known case study. The extensibility of such set will be clearer in the following sections.

The list data type  The semantic denotation of our framework assumes to use the pre-defined data type list and a set of operators on it. In particular, in a Ocaml-like fashion, the framework includes:

- hd: (list of 'a) → 'a return the head of the list
- tl: (list of 'a) → list of 'a returns the tail of the list
- len: (list of 'a) → int returns the length of the list

5.2 A type system of control primitives

Our primitives type system provides four constructors each instantiating a certain primitive. In an Ocaml-like notation, we should define type C whose all the primitives belong as:

\[
\text{type } C = \text{Fun of } (I \rightarrow A) | \\
\text{Seq of } C \text{ list } | \\
\text{Apply of } C | \\
\text{Pipe of } C \text{ list }
\]

We can design a parallel application by simply composing control patterns each embedding its own control behavior. For example, the expression

\[
\text{Pipe([Fun(f1); Apply(Fun(f2)); Seq([Fun(f3); Fun(f4)]))])}
\]

represents a pipeline of three stages: the first one is a Fun primitive embedding the sequential function f1; the second stage is an Apply primitive replicating the application of a Fun primitive that embeds the sequential function f2; the third stage is a sequential composition of two subgraphs, i.e. two Fun primitives embedding the sequential functions f3 and f4, respectively.

We have to point out that such expression represents only a pure control structure that is completely independent on which data the primitives will access and/or how data will be handled by the parallel behavior of the application. Such control structure has its own behavior given by its components, e.g. it exploits a precise semantics that will be investigated in the next chapter.

Let us show one after the other the functionality offered by each of the primitives considered in this thesis.
5.2.1 Primitive Fun

Fun is the type representing a generic sequential function. A Fun primitive is instantiated around a sequential function of type $I \rightarrow A$ that is the function the primitive will embed and execute atomically on a single processor.

It is worthwhile that the embedded function takes an iterator object as argument since in our model the iterator encapsulates not also a set of accessible values but an accessing behavior. Such behavior regards both access violation to the input data set, the order by which the values are provided to the function and the possibility to access it sequentially or in parallel.

Example 5.2.1 A graph application representing a single sequential function can be instantiate by selecting the Fun primitive as component of the graph. Its constructor requires a sequential function to be required. Let us suppose that such function, namely $f(it)$ receives an iterator $it$ as argument and returns an abstract data type built by doubling all the tasks provided by $it$. Let $\{1; 2; 3; 4; 5\}$ be such set of values. Then, given a function $f : I \rightarrow A$, the semantic expression

$$G \leftarrow \text{Fun}(f)$$

represents an application graph $G$ related to a sequential function $f$ encapsulated by the primitive Fun.

5.2.2 Primitive Seq

Seq is the type representing the sequential composition of a list of primitives given as input parameter to the constructor. Two or more primitives aggregated by a Seq instantiation are intended to be executed one after the other.

Example 5.2.2 Let $G_1 = \text{Fun}(f_1)$ and $G_2 = \text{Fun}(f_2)$ two application graph each representing a sequential function to be executed. Such graph can be evaluated sequentially by composing them into a new application graph built by the Seq primitive whose constructor requires a list of subgraph as argument. Thus, the expression

$$G_1 \leftarrow \text{Fun}(f_1)$$

represents the first subgraph, as well as

$$G_2 \leftarrow \text{Fun}(f_2)$$

represents the second one. Indeed, the global application graph $G$ is given by the expression gluing the two subgraphs into an instance of Seq, thus

$$G \leftarrow \text{Seq}([G_1; G_2])$$

5.2.3 Primitive Apply

Apply is the type representing the parallel application of an inner primitive to a set of values. The constructor requires only the primitive to be applied, while the set will be provided by an iterator given at runtime.

Example 5.2.3 Let us consider the graph $G = \text{Fun}(f)$ defined in example 5.2.1. We may apply such graph to all the elements provided by an iterator on the view of items $v = \{x_1, \ldots, x_n\}$, by embedding $G$ into an Apply instance and by instancing as iterator on $v$ a parallel iterator. Thus, the expression

$$G' \leftarrow \text{Apply(Fun}(f))$$

represents the related application graph inside the framework.

5.2.4 Primitive Pipe

Pipe represents the pipeline of a list of primitives each implementing a stage of the pipeline.
Example 5.2.4 Let us describe a graph application $G$ as a pipeline of three stages: the first stage $S_1$ is a sequential function $f_1$; the second stage $S_2$ is a parallel application of a function $f_2$ and the third stage, $S_3$ is a sequential composition of two $\text{Fun}$ primitives $f_3$ and $f_4$.

$$S_1 \leftarrow \text{Fun}(f_1)$$
$$S_2 \leftarrow \text{Apply}(\text{Fun}(f_2))$$
$$S_3 \leftarrow \text{Seq}([\text{Fun}(f_3); \text{Fun}(f_4)])$$
$$G \leftarrow \text{Pipe}([S_1; S_2; S_3])$$

5.3 Evaluation functions

The evaluation (or execution) of a graph application could be represented as a function in two variables: the nesting of primitives representing the control graph and the iterator built on top of the input data set. The result provided by the evaluation process is a new abstract data type, that is the result of all the transformations applied to the input abstract data type through a set of control primitives evaluations and related iterator settings. Thus, the evaluation function has formal type

$$\mathcal{E} : \mathcal{C} \times \mathcal{I} \rightarrow \mathcal{A} \cup \mathcal{C}$$

Example 5.3.1 Let us suppose we encode a parallel application applying to all the elements of an input data set $d = \{x_1, \ldots, x_n\}$, the same function $f$. The expected result is a data set $d' = \{f(x_1), \ldots, f(x_n)\}$ and the graph application will be an $\text{Apply}(\text{Fun}(f))$ primitive while the abstract data type implementing $d$ will be accessed, for instance, via a parallel iterator on an array view defined as follows:

$$\text{it}_p \leftarrow \text{getParIterator}(\text{getArrayView}(d))$$

The evaluation function $\mathcal{E}$ takes $\text{it}_p$ and the nested primitives as input and will return the output result $d'$ as consequence of its application, after a certain number of steps. Thus,

$$\mathcal{E}(\text{Apply}(\text{Fun}(f)), \text{it}_p) = \{f(x_1), \ldots, f(x_n)\}$$

We must point out the evaluation function will perform differently if the iterator belongs to a sequential iterator type. In fact, in case of a parallel iterator, all the values $f(x_i)$ are produced by a parallel execution of the related $\text{Fun}$ primitives. Instead, in case of a sequential iterator, the values are provided one after the other just because the iterator provides tasks sequentially to the $\text{Apply}$ primitive dispatching them.

The evaluation process The evaluation of a graph application proceeds across a finite set of steps or transformations involving both the structure of the control primitive and the current state of the iterator at hand. Each transformation is represented by an inference rule stating an equivalence relation between the left-hand and the right-hand member of the transformation. $T_1$ is a schematic example of such a rule:

$$T_1 : \frac{C(E_1)}{E_1 \xrightarrow{\mathcal{E}} E_2}$$

The associated semantics states that expression $E_1$ is transformed by rule $T_1$ into the equivalent expression $E_2$ if condition $C$ on $E_1$ holds.

At each step, $E_1$ represents the evaluation of the pair $\mathcal{C} \times \mathcal{I}$ where the states of the collective and the iterator depend on the depth at which the evaluation process is. At the end of the evaluation process, $E_2$ represents the final result, that is an output abstract data type, while in the middle of the process $E_2$ represents the configuration of $E_1$ after one evaluation step (e.g. it is still a control primitive to evaluate). Such a step involves both the interpretation of the control structure and the evaluation of operators handling iterators.
Example 5.3.2 Let Fun\( (f) \) be the application we want to evaluate. The inference rule representing the execution of its corresponding graph is written as:

\[
\frac{f : I \rightarrow A}{E(Fun(f), it) \rightarrow f(it)}
\]

The rule states that the evaluation of the graph Fun\( (f) \), coupled with the iterator \( it \) representing the input data set is equivalent to the evaluation of \( f \) applied to the input iterator. The expression \( f(it) \) is a kind of structural evolution (from a control pattern to a pure function) of the starting primitive. In this case, the step involves only the transformation of the primitive structure, not also of the iterator. As we will see in a while, the most of rules involve also the evaluation of some operators on the iterator, as for instance, current(), skip() and so on.

Regarding the evaluation of a Fun primitive, the only needed condition for applying the transformation is that the inner function \( f \) matches a precise type

Example 5.3.3 The inference rule representing the execution of the graph \( G = \text{Apply}(Fun(f)) \) coupled with a parallel iterator \( it_p \) where size\( (it_p) = n \) looks like

\[
\frac{it_p : \text{ParIterator} \land \text{hasNext}(it_p) \rightarrow \text{true}}{E(\text{Apply}(Fun(f)), it_p) \rightarrow ||_{i=1}^n E(Fun(f), it_i)}
\]

meaning that if provided that there are more elements to access, an \text{Apply} primitive is transformed into a parallel evaluation (whose operator is represented by the symbol \( || \) of the nested one, i.e. Fun\( (f) \)). Each instance of Fun\( (f) \) is coupled with an iterator \( it_i \) encapsulating the only portion of data to be accessed by the instance itself.

It should be pointed out that the evaluation of a primitive, is strict, in the sense that before evaluating it, all its arguments must be evaluated (thus, must be reduced), first. This is needed because, as mentioned before, at the end we would like to globally quantify (and qualify) the computational cost of the graph evaluation. Such cost should result as being the sum of the costs of all the transformations applied during the process. But, in order to associate a cost to a transformation, the transformation itself must be a sort of “closed”, “complete” macro-step to which a computational value can be associated in a deterministic manner.

Summarizing, \( E \) is a function that applies structural transformations to a primitive aiming at evaluating its computational semantics and getting the final data set result. But, as it can be seen in example 5.3.2, structural transformations are not the only kind of evaluation needed for executing a graph in the framework. The evaluation of functions like \( f(it) \), current\( (it) \), hasNext\( (it) \) etc. that we will see in many other inference rules need a separate semantic description. In fact, they don’t represent changes in the structure of the graph (and, as a consequence, to its control behavior): they simply represent values whose quantification requires to execute a function. In other words, they corresponds to the execution of method calls or subroutines local to the caller.

In order to describe the evaluation of a pure function (i.e. not a primitive) into the framework, we introduce a special transition labeled as \( \nu \). Each time a transformation like

\[
E_1 \xrightarrow{\nu} e
\]

occurs, it has be read as the evaluation (or execution) of the expression (a local function) \( E_1 \) that provides as result the object \( e \).

Example 5.3.4 All the operator on an iterator object are transformed by a \( \nu \) transition. As an example, let \( it = (\sigma, p) \) be a sequential iterator, then

\[
\text{size}(it) \xrightarrow{\nu} \text{size}(\sigma)
\]

Operators on the list data structure (that is better detailed in the paragraph below), are also evaluated by this transition, thus
5.4 The **builder** property

Each collective is provided with a special property whose usage is quite indispensable in the evaluation of nested primitives and it will be explained through an example.

Let us suppose to have an application structured as follows: the outer primitive is an **Apply** coupled with an iterator providing block of rows of a matrix view. The primitive nested by **Apply** (i.e. the inner primitive that must be applied to each block of rows), is another **Apply** that evaluates a certain sequential primitive **Fun**\((f)\) on the items provided by its input iterator. The question is: what type of iterator needs the inner **Apply** primitive? A sequential iterator or a parallel one? To which access pattern does it refer?

**Example 5.4.1** These questions arise also in the inference rule depicted in example 5.3.3. The rule shows that an **Apply** primitive is rewritten as the parallel evaluation of a set of **Fun**\((f)\) instances provided that **Fun**\((f)\) is the nested primitive. Each of the \(n\) instances is coupled with its own iterator \(i.t, i \in [1, n]\) encapsulating a portion of the input data. At this point, it should be understood the type of such iterator, to which access pattern it refer in order to know the access policy for each **Fun** instance.

We could imagine that the type of the inner iterator could be the same as the outer one. But this option would lead to an inflexible system of nesting.

The answers to these questions are given by the **builder** property. Such property represents an association between an iterator type and its own primitive. As it will be seen in the next chapters, the evaluation of a primitive is always preceded by the construction of the iterator type such primitive is expecting. In this way each primitive can be associated with its own type of iterator.

Now we are able to provide a definition for the property.

**Definition 5.4.1** *A builder is the property of each primitive to instantiate its own iterator.*

Functionally, a **Builder** is a function associating to each collective another function for building iterators starting from an abstract data type given as input. Formally,

\[
Builder : C \rightarrow A \rightarrow I
\]

For sake of notational simplicity we will substitute the expression \(Builder(c)\) with \(c.B\), evidencing that a builder is a property of a collective.

As mentioned before, his role is crucial for a nested collective in all those cases in which its access behavoir has to be specialized. In fact, it is worthwhile that each collective of the framework is provided with a default **Builder** that the user can specialize on the basis of his needs.

\[
\forall c \in C \text{ the default } Builder \text{ is defined as follows:}
\]

\[
c.B : A \rightarrow I \quad \text{and} \quad c.B(adt) = getSeqIterator(getArrayView(adt)) \quad (5.1)
\]

This means that, by default, each collective is intended to be coupled with (e.g. receives tasks from) a sequential iterator built from an array view (thus, a sequential view) of the values composing the input data space \(adt\).

The notation we will assume for overwriting a **builder** is the following:

\[
C_1\{B(d) = \ldots\}
\]

stating that the default builder \(C_1.B\) of the collective \(C_1\) is overwritten by the new property definition \(B(d)\).
Example 5.4.2 As an example, let us define a three stages pipeline in which the first and the third stage encapsulate the sequential function \( f_1 \) and the second one is an `Apply` of the sequential function \( f_2 \). The graph application is encoded as follows:

\[
\text{Pipe}[\text{Fun}(f_1); \text{Apply}(\text{Fun}(f_2)); \text{Fun}(f_1)]
\]

All the three collectives will access the element of the input abstract data type through a sequential iterator defined on an array view given (the one provided by their `default` builder). Let us suppose that the input abstract data type is a matrix: the second stage applies function \( f_2 \) to the single element of the matrix provided by a sequential iterator. If the user wants to change such behaviour, allowing the second stage to access row by row the input matrix, the definition of its application graph will change as follows:

\[
\text{Pipe}[\text{Fun}(f_1); \text{Apply}(\text{Fun}(f_2))\{B(s) = \text{ByRowPattern}(\text{getMatrixView}(s), 1)\}; \text{Fun}(f_1)]
\]

In this case, the second stage `Apply` will access the matrix through the iterator provided by its rewritten builder \( B(s) \), thus row by row respect to the matrix view structure of the input data.
Chapter 6

Evaluation of the application graph

In this chapter, we will detail the inference rule our semantic framework is based on to formalize the evaluation process of (a graph of) primitives. Such formalization will be provided through inference rules just to emphasize the evaluation path. Thus, the inference system is analyzed for each primitive focusing on how it is coupled with each iterator and transformed from its original definition to the final result.

6.1 Evaluation of iterators

As mentioned in Section 5.3, the evaluation of a graph application could involve the evaluation of one or more operators applied to iterators. Such operators are related to state modifications of the iterators to which they are applied and, then in the context of our framework, they can be considered pure functions. For this reason, we represent the evaluation of these operators through the function $V$.

Let us introduce an useful notation to clarify the current state of an iterator: the expression $it(t)$ will indicate that $it = (\sigma, p)$ is an iterator whose internal pointer $p$ points to the $t$-th position of $\sigma$. In other words, $\text{current}(it(t))$ returns the $t$-th element of the view accessed by the iterator $it$; $\text{hasNext}(it(t))$ returns true if and only if the size of $\sigma$ is greater than (or equal to) $t$. If $t$ is an illegal value, an execution error will arise.

Tab. 6.1 shows the inference rules related to the evaluation of iterator data type. For an informal explanation of each operator, see section 4.4.

6.2 Evaluation of primitives

Before a deep analysis of the transformation rules related to our primitives, we have to introduce a couple of semantics instruments needed to model an aspect strictly bounded with parallel computing: time. In fact we will see in the next sections that expressing a primitive evaluation, also means expressing its evolution in time.

6.2.1 Operators for handling time

Let us take into account rule $T_1$ introduced in section 5.3. The transformation $E_1 \rightarrow E_2$ represents a macro-step allowing to rewrite the expression or, better, the primitive $E_1$ into a primitive $E_2$. Such a derived primitive can be a ground primitive (for instance, it could be the pure function
Transformation rule

\[
\begin{align*}
\text{it} &= (ap, p) \land \text{size}(ap) = n \\
\text{size}(\text{it}) &\xrightarrow{v} n \\
\text{it} &= (ap, p) \land (p < \text{size}(ap)) \\
\text{hasNext}(\text{it}) &\xrightarrow{v} \text{true} \\
\text{it} &= (ap, p) \land (p \geq \text{size}(ap)) \\
\text{hasNext}(\text{it}) &\xrightarrow{v} \text{false} \\
p < \text{size}(\text{it}) &\xrightarrow{v} \text{skip}(\text{it}(p-1)) \\
\text{skip}(\text{it}(p)) &\xrightarrow{v} \text{it}(p) \\
p \geq \text{size}(\text{it}) &\xrightarrow{v} \text{skip}(\text{it}(p)) \\
\text{skip}(\text{it}(p)) &\xrightarrow{v} \bot
\end{align*}
\]

\[
\begin{align*}
\text{it} &= (ap, p) : \text{SeqIterator} \land v = \text{get}(ap, p) \in \mathcal{A} \\
\text{current}(\text{it}(p)) &\xrightarrow{v} v \\
\text{it} &= (ap, p) : \text{ParIterator} \land v_i = \text{get}(ap, i) \\
\text{current}(\text{it}) &\xrightarrow{v} \{v_i \mid \forall i \in [0, \text{size}(\text{it})]\} \\
\text{it} &= (ap, p) \land \text{it}' = \text{set}(ap, o, p) \\
\text{set}_\text{curr}(\text{it}, o) &\xrightarrow{v} \text{it}'
\end{align*}
\]

Table 6.1: Evaluation rules on the iterator data type

$f(\text{it})$ seen in example 5.3.2) or it can be a composition of primitives. The semantics operators gluing two or more primitives in time are the so called temporal operators\(^1\), each representing:

- the parallel evaluation of one or more arguments. We call such an operator \texttt{pardo} and we will denote it by means of the symbol “$||$”.

- the sequential evaluation of one or more arguments. We call such an operator \texttt{seqdo} and we will denote it by means of the symbol “;”.

Sequential execution The operator \texttt{seqdo} models the sequential evaluation of two (or more) primitives. The transformation of $(C_1; \ldots; C_n)$ implies the transformation, i.e. the evaluation, of $C_1$ coupled with the input iterator $\text{it}$ and after that, in the next macro-step, the evaluation of $C_2$ and so on. Tab.6.2 shows them.

This rule provides a first example about role and usage of the \texttt{builder} property. In fact, after the evaluation of $C_1$ and in order to evaluate $C_2$ we have to instantiate the iterator this primitive is going to use. Such iterator should be built \texttt{on-the-fly} on top of $x$, i.e. the result of the preceding primitive $C_1$. Hence, $C_2.B$ carries the factory information needed to set up the correct view on $x$.

\(^1\)The expression “temporal operators” doesn’t have any relation with the homonym type of operators known in logic programming. Here, “temporal” suggests simply that the operator expresses time.
6.2. EVALUATION OF PRIMITIVES

<table>
<thead>
<tr>
<th>Transformation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n &gt; 1 \land E(C_1, it) \rightarrow x \land C_2.B(x) \rightarrow it' )</td>
</tr>
<tr>
<td>( E(\langle C_1; \ldots; C_n \rangle, it) \rightarrow E(\langle C_2; \ldots; C_n \rangle, it') ) (6.9)</td>
</tr>
<tr>
<td>( n = 1 )</td>
</tr>
<tr>
<td>( E(\langle C_1; \ldots; C_n \rangle, it) \rightarrow E(C_1, it) ) (6.10)</td>
</tr>
</tbody>
</table>

Table 6.2: Transformation rules for reducing seqdo

and the related iterator, since it is just a function taking an abstract data type as argument and returning an associated iterator.

As mentioned in section 5.4, a default builder is associated to each collective. Such builder instantiates a sequential iterator on the input abstract data type to which an array view is applied. This means that the user have to overwrite the builder property only in those cases in which he need a different type of access method to the data space.

Evaluation of pardo Tab. 6.3 summarizes the transformation rules needed to evaluated the operator pardo. In the table, \( C_1, C_2, C_3 \in C \) and \( i_1, i_2, i_1', i_2' \in I \). In order to simplify the denotation, each rule is given expressing only two arguments but since the associative property 6.11 holds on pardo, they can be extended to a greater number of arguments.

\[(C_1 \parallel C_2) \parallel C_3 \equiv (C_1 \parallel C_2) \parallel C_3\] (6.11)

It should be pointed out that also the evaluation of pardo, as of all the other primitives, is strict, in the sense that before evaluating them, all their arguments must be reduced, first (see section 5.3)

As it can be seen, since the evaluation is strict, we need rule 6.12(6.13) for the left-hand (right-hand) reduction, in which the reduction of the left (right) argument precedes the reduction of the right (left) one, rule 6.14 for the concomitant reduction of both the arguments, and rules 6.15-6.17 a set of rule reducing the operator pardo itself. It should be pointed out that in the evaluation of a pardo expression, no assumption on the number of available processors has been taken into account. In fact, the objective of the semantics is firstly to represent the evaluation process of a graph, outlining its functional structure and, hence, its computational potentiality. Of course, the number of available processors will influence the overall performance of the application but at a level in which the actual goal of our formal system is to understand the parallel nature of the application, such information can be abstracted away.

Let us explore the evaluation of all the other collectives included into our framework. The following sections are structured in two parts: the former explaining the semantics of the collective, the latter showing its transformation rule.

6.2.2 Evaluation of Fun

Fun is a primitive encapsulating a sequential function \( f : I \rightarrow A \). The function receives an iterator it embedding the access method on the input abstract data type. The construction of such iterator can be responsibility of the programmer or can of the Fun builder. In the latter case, before the evaluation of the primitive, an instance of its expected iterator type is built by the framework by referring the property Fun.B (see section 5.4). Thus, it could be a default iterator (a SeqIterator type iterator on an array view of the input elements) or it could be re-defined by an overwriting of the primitive’ s builder property.
CHAPTER 6. EVALUATION OF THE APPLICATION GRAPH

Transformation rules

\[ C_2(it_2) \xrightarrow{\xi} x' \land C_1(B(x')) \xrightarrow{\nu} it_1 \]
\[ C_1(C_2(it_2)) \parallel C_3(it_3) \xrightarrow{\xi} C_1(it_1) \parallel C_3(it_3) \]  
(6.12)

\[ C_2(it_2) \xrightarrow{\xi} x' \land C_3.B(x') \xrightarrow{\nu} it_3 \]
\[ C_1(it_1) \parallel C_3(C_2(it_2)) \xrightarrow{\xi} C_1(it_1) \parallel C_3(it_3) \]  
(6.13)

\[ C_2(it_2) \xrightarrow{\xi} x'' \land C_1(it_1) \xrightarrow{\xi} x' \land C_2.B(x'') \xrightarrow{\nu} it_2 \land C_1.B(x') \xrightarrow{\nu} it_1 \]
\[ C_3(C_1(it_1)) \parallel C_4(C_2(it_2)) \xrightarrow{\xi} C_3(it_3) \parallel C_4(it_4) \]  
(6.14)

\[ C_1(it_1) \xrightarrow{\xi} x' \land C_2(it_2) \xrightarrow{\xi} y' \]
\[ C_1(it_1) \parallel C_2(it_2) \xrightarrow{\xi} \{x', y'\} \]  
(6.15)

\[ C_1(it'_1) \xrightarrow{\xi} x' \land C_2(it_2) \xrightarrow{\xi} C_2(it'_2) \]
\[ C_1(it_1) \parallel C_2(it_2) \xrightarrow{\xi} x' \cup C_2(it'_2) \]  
(6.16)

\[ C_1(it_1) \xrightarrow{\xi} C'_1(it'_1) \land C_2(it_2) \xrightarrow{\xi} x'' \]
\[ C_1(it_1) \parallel C_2(it_2) \xrightarrow{\xi} C'_1(it'_1) \cup x'' \]  
(6.17)

Table 6.3: Transformation rules for reducing \textit{pardo}

Functionally, the result of evaluating the \textit{Fun} primitive is the abstract data type obtained by executing \( f \). From the semantic framework’s point of view, the evaluation of \textit{Fun} is encoded by rule 6.18.

Transformation rule

\[ f : I \rightarrow A \land f(it) \xrightarrow{\nu} v \]
\[ \mathcal{E}(\text{Fun}(f), it) \rightarrow v \]  
(6.18)

Table 6.4: Transformation rule for reducing primitive \textit{Fun}.

The evaluation of a \textit{Fun} primitive is responsibility of the local evaluation function, since it corresponds to a local invocation of the function \( f \) that \textit{Fun} encapsulate.

\textbf{Example 6.2.1} Let \( f : I \rightarrow A \) defined as

\[ f(it) = \begin{cases} 
\text{append}(2 \times \text{current}(it), f(\text{skip}(it))) & \text{if hasNext(it)} \\
\{\} & \text{if not hasNext(it)} 
\end{cases} \]

where \textit{append} is the operator on abstract data types defined in Tab.4.2 and let us suppose to have instantiated a sequential iterator on an input view of items \( v = < 1; 2; 3 > \). Then, the evaluation of \( \text{Fun}(f) \) proceeds as follows:

\subsection*{6.2.3 Evaluation of Seq}

The primitive \textit{Seq} represents the evaluation of a list of encapsulated collectives or primitives that are evaluated \textit{sequentially}, each assuming as input the result provided by its predecessor in the list.
6.2. EV ALUA TION OF PRIMITIVES

\[ E(\text{Fun}(f), \text{it}) \]
\[ \rightarrow \{ f : I \rightarrow A \text{ type matching verified} \} \]
\[ f(\text{it}) \]
\[ \rightarrow \{ \text{a function evaluation is needed} \} \]
\[ [2; 4; 6] \]

The graphs of primitives sequentialized by \texttt{Seq} are listed into a list data type. Functionally, the behavior of this primitive can be represented in terms of a chain of \texttt{seqdo} application as follows:

\[ \text{Seq}([C_1, \ldots, C_n]) = C_1; C_2; \ldots; C_n = C_n(\ldots(C_1)\ldots) \]

The evaluation of \texttt{Seq} proceeds through three different transformation rules.

Rule 6.19 is related to a list of sequential primitives longer than 1: the head of the list is evaluated and after that the evaluation is recursively invoked on the tail of the list.

The end of the evaluation process is represented by rule 6.20 that is related to a list of sequential primitive whose length is equal to 1: in this case the evaluation of the primitive coincides with the evaluation of the graph heading the list.

Rule 6.21 is related to an empty list of primitives on which \texttt{Seq} is undefined.

The three rules are shown in Tab.6.5

<table>
<thead>
<tr>
<th>Transformation rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{len}(\text{lst}) \xrightarrow{\nu} n &gt; 1 \land E(\text{hd}(\text{lst}), \text{it}) \xrightarrow{\varepsilon} x )</td>
</tr>
<tr>
<td>( E(\text{Seq}(\text{lst}), \text{it}) \rightarrow E(\text{Seq}(\text{tl}(\text{lst}), \text{tl.hd}(\text{lst}).B(x))) )</td>
</tr>
<tr>
<td>( \text{len}(\text{lst}) \xrightarrow{\nu} 1 )</td>
</tr>
<tr>
<td>( E(\text{Seq}(\text{lst}), \text{it}) \rightarrow E(\text{hd}(\text{lst}), \text{it}) )</td>
</tr>
<tr>
<td>( \text{len}(\text{lst}) \xrightarrow{\nu} n \leq 0 )</td>
</tr>
<tr>
<td>( E(\text{Seq}(\text{lst}), \text{it}) \rightarrow \bot )</td>
</tr>
</tbody>
</table>

Table 6.5: Transformation rule of primitive \texttt{Seq}

**Example 6.2.2** Starting from the sequential function \( f \) defined in the example 6.2.1, we can now instantiate the graph application

\[ \text{Seq}([\text{Fun}(f); \text{Fun}(f)]) \]

as a sequential composition of two \texttt{Fun} primitives, each evaluating \( f \).

Let \( \text{it} \) be the iterator defined in the example 6.2.1 and let \( \text{lst} = [\text{Fun}(f); \text{Fun}(f)] \) be the list of primitives composing the application. The graph of primitives evaluated respect to \( \text{it} \) will be transformed by the following steps:

\[ E(\text{Seq}(\text{lst}), \text{it}) \]
\[ \rightarrow \{ \text{len}(\text{lst}) \xrightarrow{\nu} 2 \land E(\text{hd}(\text{lst})) \equiv E(\text{Fun}(f)) \wedge E(\text{Fun}(f)) \rightarrow [2; 4; 6] \} \]
\[ E(\text{Seq}(\text{tl}(\text{lst}), \text{tl.hd}.B([2; 4; 6]))) \]
\[ \equiv \{ \text{tl}(\text{lst}) = [\text{Fun}(f)] \text{ and let } \text{it'} = \text{tl.hd}.B \text{ the default iterator} \} \]
\[ E(\text{Seq}(\text{lst}, \text{it'})) \]
→ \{\text{len}(\text{lst}) \xrightarrow{\nu} 1\}

\mathcal{E}(\text{hd}(\text{lst}), \text{it})
≡\{\text{hd}(\text{lst}) = \text{Fun}(f)\}

\mathcal{E}(\text{Fun}(f), \text{it})
→ \{\text{by applying rule 6.18}\}

[4; 8; 12]

### 6.2.4 Evaluation of Apply

The constructor for the \texttt{Apply} primitive is given by

\texttt{Apply}(\mathcal{C})

The primitive applies the encapsulated primitive of type \mathcal{C} \textit{to all} the elements provided by an iterator given as input at evaluation time.

The behavior by which such application proceeds depends on the type of the iterator. If the iterator belongs to the \texttt{ParIterator} type, then \(\forall i \in [1, \text{size}(\text{it})]\) a primitive \(P_i \in \mathcal{C}\) is allocated and evaluated \textit{in parallel} with respect to the others, thus exploiting a plain data parallelism. Indeed, if the iterator belongs to the \texttt{SeqIterator} type, a primitive \(P_i\) is allocated as soon as an element is provided by the iterator accessed in a sequential manner. In this case the \texttt{Apply} primitive exploits plain task parallelism.

Let \(\text{it}\) be an iterator returning the set of items \(\{x_1, \ldots, x_n\}\), then the evaluation of \texttt{Apply} is functionally equivalent to the abstract data type obtained by the application of \(P\) to all the items provided by the iterator \(\text{it}\), thus:

\[\mathcal{E}(\text{Apply}(P), \text{it}) = \{P(P.B(x_1)), \ldots, P(P.B(x_n))\}\]

\[
\begin{array}{c}
\text{it} : \text{ParIterator} \wedge \text{hasNext}(\text{it}) \xrightarrow{\nu} \text{true} \\
\mathcal{E}(\text{Apply}(\mathcal{C}), \text{it}) \rightarrow \parallel_{i=1}^{\text{size}(\text{it})} \mathcal{E}(\mathcal{C}, \mathcal{C}.B(\text{current}(\text{it}) : i))
\end{array}
\quad (6.22)

\[
\begin{array}{c}
\text{it} : \text{SeqIterator} \wedge \text{hasNext}(\text{it}) \xrightarrow{\nu} \text{true} \\
\mathcal{E}(\text{Apply}(\mathcal{C}), \text{it}) \rightarrow \mathcal{E}(\mathcal{C}, \mathcal{C}.B(\text{current}(\text{it}))) \parallel \mathcal{E}(\text{Apply}(\mathcal{C}), \text{skip}(\text{it}))
\end{array}
\quad (6.23)

\[
\begin{array}{c}
\text{hasNext}(\text{it}) \xrightarrow{\nu} \text{false} \\
\mathcal{E}(\text{Apply}(\mathcal{C}), \text{it}) \rightarrow \{\}
\end{array}
\quad (6.24)
\]

<table>
<thead>
<tr>
<th>Transformation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{it} : \text{ParIterator} \wedge \text{hasNext}(\text{it}) \xrightarrow{\nu} \text{true}</td>
</tr>
<tr>
<td>\mathcal{E}(\text{Apply}(\mathcal{C}), \text{it}) \rightarrow \parallel_{i=1}^{\text{size}(\text{it})} \mathcal{E}(\mathcal{C}, \mathcal{C}.B(\text{current}(\text{it}) : i))</td>
</tr>
<tr>
<td>(6.22)</td>
</tr>
<tr>
<td>\text{it} : \text{SeqIterator} \wedge \text{hasNext}(\text{it}) \xrightarrow{\nu} \text{true}</td>
</tr>
<tr>
<td>\mathcal{E}(\text{Apply}(\mathcal{C}), \text{it}) \rightarrow \mathcal{E}(\mathcal{C}, \mathcal{C}.B(\text{current}(\text{it}))) \parallel \mathcal{E}(\text{Apply}(\mathcal{C}), \text{skip}(\text{it}))</td>
</tr>
<tr>
<td>(6.23)</td>
</tr>
<tr>
<td>\text{hasNext}(\text{it}) \xrightarrow{\nu} \text{false}</td>
</tr>
<tr>
<td>\mathcal{E}(\text{Apply}(\mathcal{C}), \text{it}) \rightarrow {}</td>
</tr>
<tr>
<td>(6.24)</td>
</tr>
</tbody>
</table>

Table 6.6: Transformation rule of primitive \texttt{Apply}

For a better explanation of the three rules related to the \texttt{Apply} primitive, we need to introduce the following notation:

Let \(x \in \mathcal{A}\) be an abstract data type and \(i \in [1, \text{size}(x)]\), then \(x : i\) stands for the \(i\)-th element of an array view built on the input abstract data type \(x\).
In rule 6.22, the operator \( \text{current}(it) \) returns an abstract data type possibly composed by more then one elements. The operator \( \text{current}(it) : i \) stands for the \( i \)-th element of such abstract data type.

Let us show the inference rules for the evaluation of the \text{Apply} primitive, summarizing them in Tab.6.6. As in case of a \text{Seq} primitive, we have three different cases to formalize in three different rules. Rules 6.22 and 6.23 apply when the iterator points to a not empty item set and belongs to the type \text{ParIterator} or \text{SeqIterator}, respectively. Rule 6.24 applies when there are no more elements provided by the iterator, independently from its type and returns an empty dataset.

**Example 6.2.3** Let \( f \) and \( it \) be respectively the function and the sequential iterator defined in the example 6.2.1. We can write an application that applies \( f \) in parallel to all the elements of \( it \), by simply instantiating an \text{Apply} primitive as a graph \( G \), thus having:

\[
G \leftarrow \text{Apply}(\text{Fun}(f))
\]

The evaluation phase will proceed by testing the type of the iterator: in this case it belongs to the \text{SeqIterator} type, then the \text{Fun} primitives will be instantiated one-by-one, as soon as the singletons provided by the iterator are accessed. Analytically:

\[
\begin{align*}
\mathcal{E}(\text{Apply}(\text{Fun}(f)), \text{it}) & \rightarrow \{ \text{hasNext}(\text{it}) \Rightarrow \text{true} \} \\
& \rightarrow \{ \text{Fun}(f).B(\text{current}(\text{it})) \parallel \mathcal{E}(\text{Apply}(\text{Fun}(f)), \text{skip(\text{it}))} \}
\end{align*}
\]

\[
\begin{align*}
& \rightarrow \{ \text{Fun}(f).B(\text{current}(\text{it})) \Rightarrow ([1], 1) = \text{it}_1 \land \text{skip(\text{it))} \Rightarrow ([1; 2; 3], 2) = \text{it} \} \\
& \rightarrow \{ \text{Fun}(f).\text{it}_1 \parallel \mathcal{E}(\text{Apply}(\text{Fun}(f)), \text{it}) \}
\end{align*}
\]

\[
\begin{align*}
& \rightarrow \{ \text{by applying rule 6.18 and rule 6.23} \}
\end{align*}
\]

\[
\begin{align*}
& f(\text{it}_1) \parallel \mathcal{E}(\text{Fun}(f), \text{Fun}(f).B(\text{current}(\text{it}))) \parallel \mathcal{E}(\text{Apply}(\text{Fun}(f)), \text{skip(\text{it}))} \}
\end{align*}
\]

\[
\begin{align*}
& \rightarrow \{ \text{Fun}.B(\text{current}(\text{it})) \Rightarrow ([2], 1) = \text{it}_2 \land \text{skip(\text{it))} \Rightarrow ([1; 2; 3], 3) = \text{it} \} \\
& f(\text{it}_2) \parallel \mathcal{E}(\text{Fun}(f), \text{it}_2) \parallel \mathcal{E}(\text{Apply}(\text{Fun}(f)), \text{it}) \}
\end{align*}
\]

\[
\begin{align*}
& \rightarrow \{ \text{by applying rule 6.18 and rule 6.24} \}
\end{align*}
\]

\[
\begin{align*}
& 2 \cup 4 \cup \{ f(\text{it}_3) \parallel \{ \} \} \\
& \rightarrow \{ \text{by applying rule 6.18} \}
\end{align*}
\]

\[
\begin{align*}
& 2 \cup 4 \cup \{ 6 \parallel \{ \} \}
\end{align*}
\]

\[
\begin{align*}
& \rightarrow \{ 2; 4; 6 \}
\end{align*}
\]

**Example 6.2.4** Let us suppose that \( it \in \text{ParIterator} \) is an iterator providing the same task as the example before. The evaluation of the graph application \text{Apply}(\text{Fun}(f)) \) will proceed in the following steps:

\[
\begin{align*}
\mathcal{E}(\text{Apply}(\text{Fun}(f)), \text{it}) & \rightarrow \{ \text{hasNext}(\text{it}) \Rightarrow \text{true by rule 6.22} \} \\
& \rightarrow \{ \text{current}(\text{it}) = \{ 1, 2, 3 \} \Rightarrow \forall i \in [1, 3].\text{Fun}(f).B(\text{current}(\text{it}) : i) = \text{it}_i \} \\
& \rightarrow \{ \text{Fun}(f).\text{it}_1 \parallel \mathcal{E}(\text{Fun}(f), \text{it}_1) \parallel \mathcal{E}(\text{Fun}(f), \text{it}_2) \parallel \mathcal{E}(\text{Fun}(f), \text{it}_3) \}
\end{align*}
\]

\[
\begin{align*}
& \rightarrow \{ \text{by applying in parallel rule 6.18} \}
\end{align*}
\]

\[
\begin{align*}
& f(\text{it}_1) \parallel f(\text{it}_2) \parallel f(\text{it}_3) \\
& \rightarrow \{ \forall i \in [1, 3].f(it_i) \Rightarrow 2 \ast \text{current}(it_i) \}
\end{align*}
\]

\[
\begin{align*}
& \rightarrow \{ 2; 4; 6 \}
\end{align*}
\]
### 6.2.5 Evaluation of Pipe

The primitive

\[
\text{Pipe}(\{C_1; \ldots ; C_m\})
\]

represents the application of a chain of primitives (or a composition of primitives) to all the items provided by a sequential iterator at evaluation time. Functionally, it corresponds to applying the chain of primitives \(C_1, \ldots , C_m \in C\) to each task of the input data set, thus

\[
E(\text{Pipe}(\{C_1; \ldots ; C_m\}), \{(x_1, \ldots , x_n, p)\}) = \{C_m(C_1(x_1), \ldots , C_m(x_n))\}
\]

The primitives are given as argument to the \text{Pipe} constructor through a list data type.

Tab. 6.7 summarizes all the rules needed for evaluating a \text{Pipe} primitive. The evaluation of a \text{Pipe} primitive proceeds in two dimensions, on related to the iterator size, the other is related list length. Rule 6.25 starts the evaluation of a \text{Pipe} primitive whenever the iterator size is greater than 1. In this case, the current task \(x\) provided by the iterator becomes the input data of a \text{Pipe} instantiation in which the list length is greater than one and the iterator carrying the task \(x\) has size 1. Such instantiation (rule 6.27) proceeds in only one dimension along the length of the list \(\text{lst}\).

In the meanwhile, the evaluation of the “outer” pipeline proceeds updating the state of the iterator. The transformation process terminates when the iterator provides no more task (rule 6.26) and when in the parallel branches all the lists have been consumed (rule 6.28).

In three of them it appears the abbreviation

\[
C(it) \equiv \text{hasNext}(it) \not\Rightarrow \text{true} \land \text{size}(it) \not\Rightarrow 1
\]

#### Transformation rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| (6.25) | \[
E(\text{Pipe}(\text{lst}, it)) \equiv E(\text{Pipe}(\text{lst}, \text{hd}(\text{lst}), \text{B}(x))) \| E(\text{Pipe}(\text{lst}, \text{skip}(it)) )
\]
| (6.26) | \[
E(\text{Pipe}(\text{lst}, it)) \equiv \{\}
\]
| (6.27) | \[
E(\text{Pipe}(\text{lst}, it)) \equiv E(\text{Pipe}(\text{lst}', \text{hd}(\text{lst'}), \text{B}(x)))
\]
| (6.28) | \[
E(\text{Pipe}(\text{lst}, it)) \equiv E(\text{hd}(it))
\]

#### Table 6.7: Transformation rules for evaluating primitive \text{Pipe}

**Example 6.2.5** Let \(l = [\text{Fun}(f_1); \text{Fun}(f_2)]\) be the list of sequential stages componing the pipeline we are going to evaluate. Let \(it\) be a sequential iterator on the input abstract data view \(d = \{x_1; x_2; x_3\}\). Let \(\forall i \in \{1, \text{size}(it)\}, y_{1,i} = f_1(x_i) \land y_{2,i} = f_2(y_{1,i})\). The graph application is simply built by the constructors

\[
\text{Pipe}(\{\text{Fun}(f_1); \text{Fun}(f_2)\})
\]
Now, let us take a look to how such graph will be evaluated:

\[ \mathcal{E}(\text{Pipe}(l), \text{it}) \]
\[ \rightarrow \{ \text{hasNext}(\text{it}) \rightarrow \text{true} \land \text{size}(\text{it}) \rightarrow 3 \land \text{current}(\text{it}) \rightarrow x_1 \} \]
\[ \mathcal{E}(\text{Pipe}(l), \text{hd}(l), \mathcal{B}(x_1)) \parallel \mathcal{E}(\text{Pipe}(l), \text{skip}^{(2)}(\text{it})) \]
\[ \rightarrow \{ \text{hd}(l), \mathcal{B}(x_1) \rightarrow \text{it}_{1,1} \land \text{skip}^{(2)}(\text{it}) \rightarrow \text{it}^{(2)} \} \]
\[ \mathcal{E}(\text{Pipe}(l), \text{it}_{1,1}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}^{(2)}) \]
\[ \rightarrow \{ \mathcal{E}(\text{Pipe}(l), \text{it}_{1,1}) \rightarrow \text{true} \land \text{len}(\text{it}) \rightarrow 3 \land \]
\[ \mathcal{E}(\text{hd}(l), \text{it}_{1,1}) \rightarrow \mathcal{E}(\text{fun}(l), \text{hd}(l), \mathcal{B}(y_{1,1})) \parallel \mathcal{E}(\text{Pipe}(l), \text{hd}(l), \mathcal{B}(x_2)) \parallel \]
\[ \mathcal{E}(\text{Pipe}(l), \text{skip}^{(3)}(\text{it})) \]
\[ \rightarrow \{ \text{hd}(l), \mathcal{B}(y_{1,1}) \rightarrow \text{it}_{2,1} \land \text{hd}(l), \mathcal{B}(x_2) \rightarrow \text{it}_{1,2} \} \]
\[ \mathcal{E}(\text{Pipe}^{(2)}, \text{it}_{2,1}) \parallel \mathcal{E}(\text{Pipe}^{(2)}, \text{it}_{1,2}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}^{(3)}) \]
\[ \rightarrow \{ \mathcal{E}(\text{Pipe}(l), \text{it}_{2,1}) \rightarrow \text{true} \land \text{len}(\text{it}_{2,1}) \rightarrow 1 \land \]
\[ \mathcal{E}(\text{hd}(l), \text{it}_{2,1}) \rightarrow \mathcal{E}(\text{fun}(l), \text{hd}(l), \mathcal{B}(y_{2,1})) \parallel \mathcal{E}(\text{Pipe}(l), \text{hd}(l), \mathcal{B}(x_3)) \parallel \]
\[ \mathcal{E}(\text{Pipe}(l), \text{skip}^{(3)}(\text{it})) \]
\[ \rightarrow \{ \mathcal{E}(\text{fun}(l), \text{it}_{2,1}) \rightarrow \text{y}_{2,1} \land \text{hd}(l), \mathcal{B}(y_{2,1}) \rightarrow \text{it}_{1,2} \land \]
\[ \text{hd}(l), \mathcal{B}(x_3) \rightarrow \text{it}_{1,3} \land \text{skip}^{(4)}(\text{it}) \rightarrow \text{it}^{(4)} \} \]
\[ \text{y}_{2,1} \cup \mathcal{E}(\text{Pipe}^{(2)}, \text{it}_{1,2}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}_{1,3}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}^{(4)}) \]
\[ \rightarrow \{ \mathcal{E}(\text{Pipe}(l), \text{it}_{2,2}) \rightarrow \text{true} \land \text{len}(\text{it}_{2,2}) \rightarrow 1 \land \]
\[ \mathcal{E}(\text{hd}(l), \text{it}_{1,2}) \rightarrow \mathcal{E}(\text{fun}(l), \text{hd}(l), \mathcal{B}(y_{1,3})) \parallel \}
\[ \text{y}_{2,1} \cup \mathcal{E}(\text{fun}(l), \text{it}_{2,2}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}_{1,3}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}^{(4)}) \]
\[ \rightarrow \{ \mathcal{E}(\text{fun}(l), \text{it}_{2,2}) \rightarrow \text{false} \}
\[ \text{y}_{2,1} \cup \mathcal{E}(\text{fun}(l), \text{it}_{2,2}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}_{1,3}) \parallel \}
\[ \text{y}_{2,1} \cup \mathcal{E}(\text{Pipe}(l), \text{it}_{1,3}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}^{(4)}) \]
\[ \rightarrow \{ \mathcal{E}(\text{Pipe}(l), \text{it}_{2,3}) \rightarrow \text{false} \}
\[ \text{y}_{2,1} \cup \mathcal{E}(\text{Pipe}(l), \text{it}_{2,3}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}_{1,3}) \parallel \}
\[ \text{y}_{2,1} \cup \mathcal{E}(\text{Pipe}(l), \text{it}_{2,3}) \parallel \mathcal{E}(\text{Pipe}(l), \text{it}^{(4)}) \]
\[ \rightarrow \{ \text{y}_{2,1} \land \text{y}_{2,2} \land \mathcal{E}(\text{fun}(l), \text{it}_{2,3}) \}
\[ \text{y}_{2,1} \land \text{y}_{2,2} \land \mathcal{E}(\text{fun}(l), \text{it}_{2,3}) \]
Chapter 7

Computational costs system and rewriting rules

In this chapter we will introduce a computational costs system related to the set of formal primitives detailed in the previous chapters. Such system associates a computational cost to each macro-step involved in the transformation process. In this way the cost of an application results in the sum of the partial costs of the macro-steps needed to transform the user application into the final result. The main idea is to have a formal framework able to “suggest” structural improvements to the user application graph. Such improvements depends on the computational costs exploited by the graph structure that the framework will estimate. Obviously, the framework doesn’t need to evaluate all the graph structure to determine its global computational cost, first. The aim of this chapter is focused on defining some closed formulae that given a graph structure, is able to estimate its computational cost. In the next sections we will just demonstrate how we can obtain such formulae through the set of inference rules outlined in the previous chapter. Consequently, in chapter 10 some simple experiments will demonstrate the actual applicability of the approach through real programming examples.

7.1 A computational cost system

As mentioned in chapter 3, the goal of describing our programming model by means of a formal semantics is to have of a clear, unambiguous description of both abstract mechanisms and control primitives. On the other hand, a rigorous description of the graph evaluation process, as formalized in chapter 4, allows to establish if two graphs are equivalent, that is if two structurally different graphs evaluate the same function.

The further step promised by our framework is the ability to estimate the cost of our rewriting mechanisms. Such estimation is given through a computational cost system assigning to a given application graph $G$ a computational cost. Once the transformation of a graph $G$ into an equivalent graph $G'$ (or into the final data result) has been completed in one or more steps, the total cost of rewriting $G$ is given by adding the cost of all the intermediate steps involved in the process.

Since each application can be evaluated through this cost system, two or more functionally equivalent graphs representing the application can be compared in order to establish the cheapest in terms of computational step.

Our framework aims at using such computational cost system to automatically establish if the user application graph can be improved through a set of transformations, thus rewriting it in order to reach the better performance. Since some of such transformations have been formally proved a \textit{a priori} (i.e. statically) by our semantics framework, they can be imported in the runtime system of the implementation framework in order to make possible some preliminary optimizations on the user graph. Possibly, the set of optimizations can be extended by new static transformation rules that in the next future we will investigate. In this thesis we have considered just a set of
transformations that are well-known in literature in order to prove the feasibility of the approach and the expressive power of the costs system.

The description of a computational cost system associated to the semantics introduced in the previous chapter is the first goal of this chapter. We will describe the cost related to the evaluation of each primitive by simply labeling each transformation with a computational cost that depends on the cost of its components.

Let us suppose to have a rule $T_0$ such that

$$T_0 : \frac{C(E_1)}{E_1 \xrightarrow{f(C)} E_2}$$

The relation $\rightarrow$ has been labeled by the cost $f(C)$ payed for transforming expression $E_1$ into expression $E_2$. Such cost depends from (is a function of) the cost of evaluating the condition $C(E_1)$ that may require other preliminary reduction\(^1\). For example, if $E_1 \equiv \text{hasNext}(it)$ and $E_2 \equiv \text{true}$, then $c \equiv c_v$ where $c_v$ is the cost of the function $\mathcal{V}$, evaluating local computations. In this case $f(c) \equiv c$ because there are no more conditions to evaluate, i.e. we have not other computational costs to add.

Generally speaking, all costs that sequentially appear in the precondition of a rule become operands of the sum evaluating the total cost of the rule itself. The only exception is given by the pardo operator whose cost is the maximum afforded to reduce all its argument.

In the following we will give the cost related to the evaluation of each primitive, temporal operators and operators on iterators. We will provide an example on how to estimate the cost of a sample application and, at the end, we will give some transformation.

As mentioned above, the following implication holds:

$$E_1 \xrightarrow{\mathcal{V}} E_2 \Rightarrow E_1 \xrightarrow{c_v} E_2$$

or, in other words, all the transformations labeled by the $\mathcal{V}$ transition have a constant cost $c_v$.

From a denotational point of view we will call

$$\mathcal{S} : (\mathcal{C} \times I) \rightarrow \mathbb{N}$$

the function assigning a computational cost to an input collective coupled with a sequential or parallel iterator. However, we have to outline that

- we don’t make any assumption of the number of available processors: as mentioned in the previous chapter, the semantics is devoted to abstract away from any low level or implementation detail, rather focusing on the parallel behavior of the application graph;

- we won’t give a numerical value to the costs labeling the transformation: again, in order to quantify such costs we should refer some target architecture. Instead, our aim is to reason about the program structure from a formal as well as qualitative point of view. As we will see in the next sections, we will able to compare (potential) programs performances in terms of costs of transformation steps, without assigning numerical values.

\(^1\)We recall that the evaluation of a primitive has to be strict (see section 5.3)
7.1. A COMPUTATIONAL COST SYSTEM

7.1.1 Cost of _pardo_

As we have seen in 6.2, evaluating a _pardo_ expression means evaluating in parallel all its arguments, that is its encapsulated primitives. We have only seen that the evaluation of _pardo_ is strict, meaning that before reducing it, all its arguments must be reduced. These two conditions give the basis for the computational cost of _pardo_.

In fact, if the arguments are both reduced (rule 7.4) or they have both to be reduced from the same level (rules 7.3, 7.5 and 7.6), then the maximum reduction cost between their ones represents the _pardo_ reduction cost.

Instead, if the arguments are not reduced at the same level, that is some of the arguments needs to be reduced before the others (rules 7.1 and 7.2), than the maximum cost for reducing them represents the cost for reducing the _pardo_ expression.

Tab. 7.1 summarizes the transformation rules related to _pardo_ operator labeled by the computational cost for each step.

<table>
<thead>
<tr>
<th>Transformation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ C_2(it_2) \xrightarrow{c} x' \land C_1.B(x') \xrightarrow{c} it_1 ]</td>
</tr>
<tr>
<td>[ C_1(C_2(it_2)) \parallel C_3(it_3) \xrightarrow{c+c_1} C_1(it_1) \parallel C_3(it_3) ]</td>
</tr>
<tr>
<td>[ \frac{C_1(it_1) \parallel C_3(C_2(it_2)) \xrightarrow{c+c_1} C_1(it_1) \parallel C_3(it_3)}{C_1(it_1) \parallel C_3(it_1) \parallel C_3(it_3) \xrightarrow{\max{c_1,c_2}} C_3(C_3(it_1)) \parallel C_3(it_3)\xrightarrow{c+c_1}} ]</td>
</tr>
<tr>
<td>[ \frac{C_1(it_1) \xrightarrow{c} x' \land C_2(it_2) \xrightarrow{c} x''}{C_1(it_1) \parallel C_2(it_2) \xrightarrow{\max{c_1,c_2}} C_3(it_2) \parallel C_3(it_2)\xrightarrow{c+c_1}} ]</td>
</tr>
<tr>
<td>[ \frac{C_1(it_1) \xrightarrow{c} x' \land C_2(it_2) \xrightarrow{c} y'}{C_1(it_1) \parallel C_2(it_2) \xrightarrow{\max{c_1,c_2}} x' \cup y'} ]</td>
</tr>
<tr>
<td>[ \frac{C_1(it_1) \parallel C_2(it_2) \xrightarrow{\max{c_1,c_2}} x' \cup C_2(it_2)}{C_1(it_1) \parallel C_2(it_2) \xrightarrow{\max{c_1,c_2}} C_1(it_1) \parallel x''} ]</td>
</tr>
<tr>
<td>[ \frac{C_1(it_1) \xrightarrow{c} C_1(it_1') \land C_2(it_2) \xrightarrow{c} x''}{C_1(it_1) \parallel C_2(it_2) \xrightarrow{\max{c_1,c_2}} C_1(it_1') \cup x''} ]</td>
</tr>
</tbody>
</table>

Table 7.1: Cost system for _pardo_

7.1.2 Cost of _seqdo_

It is not surprising that the cost of evaluating a list of primitives reduced sequentially by a _seqdo_ temporal operator, is the sum of the costs payed for evaluating each of them, as depicted in Tab. 7.2.

7.1.3 Iterator costs

Since the evaluation of an iterator requires the evaluation of one or more operators on its state, there is not an actual transformation, from a structural point of view. Indeed, there is a computational evaluation and for this reason all the operators on the iterator type, are evaluated through the _ν_
### Transformation rules

\[
\begin{align*}
\text{Table 7.2: Cost system for reducing seqdo} \\
& \text{function. This means that the computational cost of each operator is equal to } c_v, \text{ that is the cost of the evaluation function } V. \\
\end{align*}
\]

\[
\begin{align*}
\text{Transformation rules} \\
& \quad n > 1 \land E(C_1, it) \xrightarrow{c_v} x \land C_2, B(x) \xrightarrow{c_v} it' \\
& E((C_1; \ldots; C_n), it) \xrightarrow{c_1 + c_v} E((C_2, \ldots; C_n), it') \\
& n = 1 \\
& E((C_1; \ldots; C_n), it) \xrightarrow{0} E(C_1, it) \\
\end{align*}
\]

### Table 7.3: Cost system for iterator type

\[
\begin{align*}
& \text{Transformation rules} \\
& \quad \text{it} = (ap, p) \land \text{size(ap)} = n \\
& \quad \text{size(it)} \xrightarrow{c_v} n \\
& \quad \text{it} = (ap, p) \land (p < \text{size(ap)}) \\
& \quad \text{hasNext(it)} \xrightarrow{c_v} \text{true} \\
& \quad \text{it} = (ap, p) \land (p \geq \text{size(ap)}) \\
& \quad \text{hasNext(it)} \xrightarrow{c_v} \text{false} \\
& \quad p < \text{size(it)} \\
& \quad \text{skip(it}^{(p-1)}) \xrightarrow{c_v} \text{it}^{(p)} \\
& \quad p \geq \text{size(it)} \\
& \quad \text{skip(it}^{(p)}) \xrightarrow{0} \bot \\
& \quad \text{it} = (ap, p) : \text{SeqIterator} \land v = \text{get(ap, p)} \in A \\
& \quad \text{current(it}^{(p)}) \xrightarrow{c_v} v \\
& \quad \text{it} = (ap, p) : \text{ParIterator} \land v_i = \text{get(ap, i)} \\
& \quad \text{current(it)} \xrightarrow{c_v} \{ v_i \mid \forall i \in [0, \text{size(it)}] \} \\
& \quad \text{it} = (ap, p) \land \text{it}' = (\text{set(ap, o, p), p}) \\
& \quad \text{set_curr(it, o)} \xrightarrow{c_v} \text{it}' \\
\end{align*}
\]

### 7.1.4 Fun cost

Functionally, the result of evaluating the Fun primitive is the abstract data type obtained by executing \( f \) on the underlying processor. Thus, the cost of such evaluation is equal to the cost of executing \( f \), a function that takes the input iterator as argument and, possibly, manipulates the provided tasks. In rule 7.15 in Tab.7.4, such cost is labeled as \( f_c \).

As an example, if \( f \) implements a function that applies a certain transformation \( T \) to each element provided by the iterator, by paying a cost \( c_T \), then \( f_c = \text{size(it)} \times c_T \).

**Example 7.1.1** Let \( f : \mathcal{I} \rightarrow \mathcal{A} \) be the function defined in the example 6.2.1. Its computational cost can be evaluated step-by-step as follows:
7.1. A COMPUTATIONAL COST SYSTEM

### Transformation rules

\[
\begin{align*}
& f : I \rightarrow A \land f(it) \rightarrow^{f_c} x \\
& \mathcal{E}(\text{Fun}(f), it) \rightarrow^{f_c} x \\
\end{align*}
\]

(7.15)

<table>
<thead>
<tr>
<th>Table 7.4: Cost system for primitive Fun</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{E}(\text{Fun}(f), it)$</td>
</tr>
<tr>
<td>$\rightarrow { f : I \rightarrow A \text{ type matching} }$</td>
</tr>
<tr>
<td>$f(it)$</td>
</tr>
<tr>
<td>$\rightarrow^{f_c} { \text{a function evaluation is needed} }$</td>
</tr>
<tr>
<td>[2; 4; 6]</td>
</tr>
</tbody>
</table>

Then the cost for evaluating Fun$(f)$ is represented by

\[
\$(\text{Fun}(f), it) = f_c
\]

(7.16)

7.1.5 Apply cost

The way an Apply primitive is evaluated depends on the type of iterator with which it is coupled but, at first, the initial cost needed for transforming Apply is equal to the cost of asking the iterator for the availability of new tasks. This starting costs are shown in Tab. 7.5.

<table>
<thead>
<tr>
<th>Table 7.5: Cost system for the Apply primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$it : ParIterator \land hasNext(it) \subseteq true$</td>
</tr>
<tr>
<td>$\mathcal{E}(\text{Apply}(C), it) \subseteq |_{i=1}^{\text{size}(it)} \mathcal{E}(C, C.B(\text{current}(it) : i))$</td>
</tr>
<tr>
<td>(7.17)</td>
</tr>
<tr>
<td>$it : SeqIterator \land hasNext(it) \subseteq true$</td>
</tr>
<tr>
<td>$\mathcal{E}(\text{Apply}(C), it) \subseteq \mathcal{E}(C, C.B(\text{current}(it))) | \mathcal{E}(\text{Apply}(C), \text{skip}(it))$</td>
</tr>
<tr>
<td>(7.18)</td>
</tr>
<tr>
<td>$hasNext(it) \subseteq false$</td>
</tr>
<tr>
<td>$\mathcal{E}(\text{Apply}(C), it) \subseteq []$</td>
</tr>
<tr>
<td>(7.19)</td>
</tr>
</tbody>
</table>

As soon as the transformation proceeds, the total cost of evaluating Apply increases by adding further costs, actually depending on the type of the iterator. Let us explain the two cases.

**Parallel iterator** Let $it \in ParIterator$ be a parallel iterator on a list of tasks $\{x_1, \ldots, x_n\}$, let $\mathcal{E}(\text{Apply}(C), it) = \{y_1, \ldots, y_n\}$ be the expected result and let

$\$(C, C.B(s)) = f_c$

the cost of the inner collective C.

By analyzing the transformation process step by step as shown it Tab. 7.6 we are going to give a close formula related to the cost of the Apply primitive.

The final cost is equal to

\[
\$(\text{Apply}(C), it) = 3c_v + c_b + \max_{i \in [1, n]} \{f_{c_i}\}
\]

(7.20)
\[ E(Apply(C), it) \]

\[ \xrightarrow{c_b} \{ \text{by applying rule 7.17} \} \]

\[ \xrightarrow{|size(it)| \atop \sum_{i=1}^{n}} E(C, C.B(current(it) : i)) \]

\[ \xrightarrow{c_v} \{ \text{since } current(it) \xrightarrow{c_v} x \} \]

\[ \xrightarrow{|size(it)| \atop \sum_{i=1}^{n}} E(C, C.B(x : i)) \]

where \( n = size(it) \). Moreover, since on a cluster of workstations it is presumed that \( c_v \approx 0 \) (because it represents the cost of evaluating a local function evaluation), and that \( f_c \) are quite close values that may be approximated with \( f_c \), we can simplify the formulae by writing:

\[ \$E(Apply(C), it) = c_b + f_c \]

**Sequential iterator**

Let \( it : SeqIterator \) a sequential iterator on the list of item \( \{x_1, \ldots, x_n\} \),

\[ E(Apply(C), it) = \{y_1, \ldots, y_n\} \]

be the expected result and \( \$E(C, C.B(s)) = f_c \).

Then the evaluation of \( Apply \) proceeds as shown in Tab.7.7:

\[ E(Apply(C), it) \]

\[ \xrightarrow{c_b} \{ \text{hasNext(it) } \xrightarrow{c_v} \text{ true} \} \]

\[ E(C, C.B(current(it))) \parallel E(Apply(C), skip(it)) \]

\[ \xrightarrow{c_v} \{ \text{current(it) } \xrightarrow{c_v} x \land skip(it) \xrightarrow{c_v} it(2) \} \]

\[ E(C, C.B(x)) \parallel E(Apply(C), it(2)) \]

\[ \xrightarrow{c_b} \{ C.B(x) \xrightarrow{c_b} it_1 \} \]

\[ E(C, it_1) \parallel E(Apply(C), it^2) \]

\[ \xrightarrow{-c_1} \{ \text{let } E(C, it_1) \xrightarrow{-c_1} y_1 \land hasNext(it(2)) \xrightarrow{c_v} \text{ true} \land c_1 \geq c_v \} \]

\[ \{ y_1 \parallel E(C, C.B(current(it(2)))) \parallel E(Apply(C), skip(it(2))) \} \]

\[ \cdots \]

\[ \xrightarrow{c_b} \{ \text{since } current(it(n)) \xrightarrow{c_v} x \land skip(it(n)) \rightarrow it(n+1) \} \]

\[ E(C, C.B(x)) \parallel E(Apply(C), it(n+1)) \]

\[ \xrightarrow{c_b} \{ C.B(x) \xrightarrow{c_b} it_n \} \]

\[ E(C, it_n) \parallel E(Apply(it(n+1))) \]

\[ \xrightarrow{-c_1} \{ E(C, it_n) \xrightarrow{-c_1} y_n \land hasNext(it(n+1)) \xrightarrow{c_v} \text{ false} \land c_v \leq c_n \} \]

\[ \{ y_1 \parallel \cdots \parallel y_{n-1} \parallel E(C, C.B(current(it(n)))) \parallel E(Apply(C), skip(it^n)) \} \]

\[ \cdots \]

\[ \xrightarrow{c_b} \{ \text{since } current(it(n)) \xrightarrow{c_v} x \land skip(it(n)) \rightarrow it(n+1) \} \]

\[ E(C, C.B(x)) \parallel E(Apply(C), it(n+1)) \]

\[ \xrightarrow{c_b} \{ C.B(x) \xrightarrow{c_b} it_n \} \]

\[ E(C, it_n) \parallel E(Apply(it(n+1))) \]

\[ \xrightarrow{-c_1} \{ E(C, it_n) \xrightarrow{-c_1} y_n \land hasNext(it(n+1)) \xrightarrow{c_v} \text{ false} \land c_v \leq c_n \} \]

\[ \{ y_1 \parallel \cdots \parallel y_{n-1} \parallel y_n \parallel \} \]

**Table 7.6:** Transformation steps for \( Apply \) given a parallel iterator

**Table 7.7:** Transformation steps for \( Apply \) given a sequential iterator
Rule 7.18 imposes a startup cost equal to \( c_v \) and, as long as the iterator provides tasks to be computed, i.e. \( \forall i \in [1, size(it)] \), a cycle of costs equal to \( c_v + c_b + c_i \) is repeated. Such cycle includes the evaluation of the iterator state, the creation of the iterator on the \( i \)-th emitted task and the evaluation of the \( i \)-th primitive. Thus,

\[
\$(Apply(C), it) = c_v + n \times \sum_{i=1}^{size(it)} c_v + c_b + c_i = c_v \times (n + 1) + c_b \times n + \sum_{i=1}^{size(it)} c_i \quad (7.21)
\]

As it can be seen by comparing 7.21 and 7.20, the type of the iterator heavily influences the cost of the application graph. In the first case it is proportional to the cost of computing a single task provided by the iterator, in the second case it is proportional to the size of the input iterator.

### 7.1.6 Seq cost

Tab.7.8 summarizes the costs related to the evaluation of a Seq primitive. As in the case of Apply, we need again to further analyze the evaluation process in order to get a formula indicating the cost of such primitive.

<table>
<thead>
<tr>
<th>Transformation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{len}(\text{lst}) \xrightarrow{c_v} n \land n &gt; 1 \land \text{E}(\text{hd}(\text{lst}), it) \rightarrow c_v \land \text{tl}(\text{lst}) \xrightarrow{c_v} \text{lst}' )</td>
</tr>
<tr>
<td>( \text{E}(\text{Seq}(\text{lst}), it) \rightarrow c_v + 3c_v ) ( \text{E}(\text{Seq}(\text{lst}'), \text{hd}(\text{lst}').B(x)) )</td>
</tr>
<tr>
<td>( \text{len}(\text{lst}) \xrightarrow{c_v} 1 )</td>
</tr>
<tr>
<td>( \text{E}(\text{Seq}(\text{lst}), it) \xrightarrow{c_v} \text{E}(\text{hd}(\text{lst}), it) )</td>
</tr>
</tbody>
</table>

Table 7.8: Computational costs for primitive Seq

Let \( it : SeqIterator \) be an iterator on the set of items \( x_1, \ldots, x_n \). The cost for evaluating the list collectives \( l = [C_1; \ldots; C_m] \) is given by the following chain of transformations:

\[
\text{E}(\text{Seq}(l), it)
\]

\[
\xrightarrow{3c_v + c_b} \{ \text{len}(l) \xrightarrow{c_v} n > 1 \land \text{E}(\text{hd}(l), it) \xrightarrow{c_v + c_v} x \land \text{tl}(l) \xrightarrow{c_v} l_2 \} \]

\[
\xrightarrow{c_v + c_b} \{ \text{hd}(l_2).B(x) \xrightarrow{c_v + c_v} it_2 \} \]

\[
\xrightarrow{c_v} \{ \text{Seq}(l_2), it_2 \} \]

\[
\xrightarrow{c_v} \{ \text{Seq}(l_n), it_n \} \]

\[
\xrightarrow{c_v} \{ \text{len}(l_n) \xrightarrow{c_v} 1 \} \]

\[
\xrightarrow{c_v} \{ \text{hd}(l_n), it_n \} \]

\[
\xrightarrow{c_v} \{ \text{C}_n, it_n \} \]

\[
\xrightarrow{c_v} \{ y_1; \ldots; y_n \} \]

In this case \( \forall i \in [1, n - 1] \) we have a constant cost for each primitive \( C_i \) in the list that is \( 3c_v + c_i + c_v + c_b \). The evaluation of the last primitive costs \( 2c_v + c_n \). Then, summarizing:

\[
\$(\text{Seq}(l), it) = (2c_v + c_n) + \sum_{i=1}^{n-1} (4c_v + c_i + c_b)
\]
\[ (n - 1) \times c_b + 2c_v \times (2n - 1) + \sum_{i=1}^{n} c_i \]

7.1.7 Pipe cost

Evaluating the cost of a Pipe of \( m \) stages requires to separately analyze the three phases in which the life cycle of this primitive is divided: startup, steady state and finalizing phase.

During the startup phase, the set of stages composing the pipeline is incrementally filled up by new tasks entering into the flow of control. This phase lasts \( m - 1 \) intervals, that is the number of tasks needed in order to have all the stages of the pipeline working.

After the startup phase, there is a steady state phase called full pipeline. During this phase, all the stages are working on (or are waiting for) a task.

At the end of the steady state phase, there is the finalizing phase, that is the phase in which the stages of the pipeline stop working, one after the other.

Table 7.9 summarizes the cost for the evaluation of a Pipe primitive. The cost of the condition

\[ C(it) = size(it) \rightarrow 1 \land hasNext(it) \rightarrow true \]

occurring in rules 7.26 and 7.27 is equal to \( c_C = 2c_v \).

<table>
<thead>
<tr>
<th>Transformation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{hasNext(it) } \rightarrow \text{true } \land size(it) \rightarrow n &gt; 1 \land current(it) \rightarrow x )</td>
</tr>
<tr>
<td>( \text{hasNext(it) } \rightarrow \text{false} )</td>
</tr>
</tbody>
</table>
| \( \text{C(it) } \rightarrow true \land \text{len(lst) } \rightarrow n > 1 \land \mathcal{E}(\text{hd(lst, it)} \rightarrow x \land tl(lst) \rightarrow \text{lst'} \) | \( \mathcal{E}(\text{Pipe(lst, it) } \rightarrow 5c_v + c \mathcal{E}(\text{Pipe(lst'}, \text{hd(lst'),B(x)})) \quad (7.26) \)
| \( \text{C(it) } \rightarrow true \land \text{len(lst) } \rightarrow n > 1 \land \mathcal{E}(\text{hd(lst, it)} \rightarrow x \land \text{C} \land \mathcal{E}(\text{C, it} \rightarrow y \) | \( \mathcal{E}(\text{Pipe(lst, it}) \rightarrow 4c_v + c \rightarrow y \quad (7.27) \)

Table 7.9: Cost system for the primitive Pipe

In the analysis of the computational cost we will use the following notation: \( it_{x,y} \) is the iterator built upon the \( y \)-th task and it encapsulates value to provide as argument to the \( x \)-th function.

Let \( it \) be a sequential iterator on the input item set \( \{x_1, \ldots, x_n\} \).

Let \( l = [C_1; \ldots; C_m] \) the list of pipeline stages.

**Startup phase**  The startup cost \( c_{\text{init}} \) is given by the first transformation step

\[ \mathcal{E}(\text{Pipe(l, it}) \]

\[ \rightarrow 3c_v \{ \text{hasNext(it) } \rightarrow \text{true } \land size(it) \rightarrow n > 1 \land current(it) \rightarrow x \} \]

\[ \mathcal{E}(\text{Pipe(l, hd,B(x)})) \| \mathcal{E}(\text{Pipe(l, skip(it)}) \]
and its value is equal to $3c_v$, such that

$$
\$ (\text{Pipe}(l), it) = c_{\text{init}} + c_{\text{reg}} = 3c_v + c_{\text{reg}}
$$

Let us determine the value of the steady state and the finalizing phase.

$\forall t \in [1, m]$ we have the cycle of transformations:

$$
\begin{align*}
\mathcal{E}(\text{Pipe}(l^{(t)}), hd(l^{(t)}).B(y_{t-1}, j)) & \quad \text{\ldots} \quad \mathcal{E}(\text{Pipe}(l), hd(l).B(x_t)) \quad \mathcal{E}(\text{Pipe}(l), \text{skip}^{(t+1)}(it)) \\
\text{\ldots} & \quad \mathcal{E}(\text{Pipe}(l^{(t)}), it_{t,1}) \quad \text{\ldots} \quad \mathcal{E}(\text{Pipe}(l^{(t)}), it_{t,1}^{(t)}) \quad \mathcal{E}(\text{Pipe}(l^{(t)}), it_{t,1}^{(t+1)})
\end{align*}
$$

We have $m - 1$ transformations of this type and they coincide with the startup phase. Thus, the cost of the startup phase is equal to

$$
c_{\text{startup}} = \sum_{t=1}^{m-1} c_b + c_v + \max_i \{5c_v + c_f_i\} = \sum_{t=1}^{m-1} c_b + 6c_v + \max_i \{c_f_i\} = (m - 1) \times (c_b + 6c_v) + \sum_{t=1}^{m-1} \max_i \{c_f_i\}
$$

**Steady state** At the $m$-th transformation, e.g. when all the stages are busy and we are going to produce the first result, let’s say $y_1$, we have:

$$
\begin{align*}
\mathcal{E}(\text{Pipe}(l^{(m)}), hd(l^{(m)}).B(y_{m-1}, j)) & \quad \mathcal{E}(\text{Pipe}(l^{(m-1)}), hd(l^{(m-1)}).B(y_{m-2}, j)) \quad \mathcal{E}(\text{Pipe}(l), \text{skip}^{(m+1)}(it)) \\
\text{\ldots} & \quad \mathcal{E}(\text{Pipe}(l^{(m)}), it_{m,1}^{(m)}) \quad \mathcal{E}(\text{Pipe}(l^{(m-1)}), it_{m-1,2}^{(m)}) \quad \text{\ldots} \quad \mathcal{E}(\text{Pipe}(l), it_{m+1}^{(m)})
\end{align*}
$$

The $m$-th step costs $6c_v + c_b + \max_i \{c_f_i\}$ and the same steps are repeated for $n - m + 1$ steps, e.g. they last as well as the steady state phase, thus costing:

$$
c_{\text{full}} = \sum_{t=m}^{n} c_v + c_b + \max_i \{4c_v + c_f_i, 5c_v + c_f_i, \forall i \in [1, m]\}
$$

**Finalizing phase** At step $n$, we have already produced $n - m - 1$ results and the pipeline starts its finalizing phase that will last $m - 1$ steps. These steps will have the following configuration

$$
\begin{align*}
y_1 & \quad \text{\ldots} \quad y_{n-m-1} \quad \mathcal{E}(\text{Pipe}(l^{(m)}), hd(l^{(m)}).B(y_{m-1}, j)) \quad \mathcal{E}(\text{Pipe}(l), hd(l).B(x_n)) \quad \mathcal{E}(\text{Pipe}(l), \text{skip}^{(m+1)}(it)) \\
\text{\ldots} & \quad \mathcal{E}(\text{Pipe}(l), hd(l).B(x_{n-1}, j)) \quad \mathcal{E}(\text{Pipe}(l), \text{skip}^{(m)}(it))
\end{align*}
$$
The cost of each step is equal to the ones occurring in the steady state, but the number of stages decrease, step after step. Hence, for each step \( t \in [n, n + m - 1] \) the computational cost is

\[
\begin{align*}
\textit{\textit{c}_{empty}} &= \sum_{t=n+1}^{n+m-1} c_v + c_b + \max\{4c_v + c_{f_m}, 5c_v + c_f, c_{f_t}, c_{f_i}, c_{f_{n+1}}, c_{f_{m+1}}, c_{f_{m+2}}\} \\
&= 3c_v + c_{\text{startup}} + c_{\text{full}} + c_{empty}
\end{align*}
\]

Summarizing, the cost of the pipeline is:

\[
\begin{align*}
\$[\mathcal{E}(\text{Pipe}(l, it)) &= 3c_v + c_{\text{startup}} + c_{\text{full}} + c_{\text{empty}} \\
&= 3c_v + \sum_{t=n}^{n+m-1} c_v + c_b + \max\{4c_v + c_{f_m}, 5c_v + c_f, c_{f_t}, c_{f_i}, c_{f_{n+1}}, c_{f_{m+1}}, c_{f_{m+2}}\} \\
&= 3c_v + \sum_{t=n}^{n+m-1} c_v + c_b + \sum_{t=n}^{n+m-1} \max\{4c_v + c_{f_m}, 5c_v + c_f, c_{f_i}, c_{f_{n+1}}, c_{f_{m+1}}, c_{f_{m+2}}\} \\
&= 3c_v + \sum_{t=n}^{n+m-1} \max\{4c_v + c_{f_m}, 5c_v + c_f, c_{f_i}, c_{f_{n+1}}, c_{f_{m+1}}, c_{f_{m+2}}\}
\end{align*}
\]

Example 7.1.2 Let \( n = 5 \) and \( m = 3 \), and let \( f \) be the function for all the stages. Then, the cost of the final pipeline application is:

\[
\begin{align*}
\$[\mathcal{E}(\text{Pipe}(l, it)) &= 3c_v + \sum_{t=n}^{n+m-1} \max\{4c_v + c_{f_m}, 5c_v + c_f, c_{f_i}, c_{f_{n+1}}, c_{f_{m+1}}, c_{f_{m+2}}\} \\
&= 3c_v + \sum_{t=n}^{n+m-1} \max\{4c_v + c_{f_m}, 5c_v + c_f, c_{f_i}, c_{f_{n+1}}, c_{f_{m+1}}, c_{f_{m+2}}\}
\end{align*}
\]
7.1.8 Summary of costs

Table 7.10 summarizes the computational cost provided in the previous pages.

<table>
<thead>
<tr>
<th>Application graph</th>
<th>Cost expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{E}(\text{Fun}(f), \text{it}_s) = \text{Fun}(f, \text{it}_p)$</td>
<td>$c_f$</td>
</tr>
<tr>
<td>$\mathcal{E}(\text{Seq}(l), \text{it}_s) = \text{Seq}(l, \text{it}_p)$</td>
<td>$(m - 1) \times c_b + 2c_v \times (2m - 1) + \sum_{t=1}^{m} c_i$</td>
</tr>
<tr>
<td>$\mathcal{E}(\text{Apply}(C), \text{it}_p)$</td>
<td>$\max_{i \in [1..n]} {c_b} + c_C$</td>
</tr>
<tr>
<td>$\mathcal{E}(\text{Apply}(C), \text{it}_s)$</td>
<td>$c_v \times (n + 1) + c_b \times n + \sum_{t=1}^{n} c_f, 3c_v +$</td>
</tr>
<tr>
<td>$\mathcal{E}(\text{Pipe}(l), \text{it}_s)$</td>
<td>$+(m - 1)(c_b + 6c_v) + \sum_{t=1}^{m-1} \max_{i \in [1..t]} {c_f}$</td>
</tr>
<tr>
<td>$\mathcal{E}(\text{Pipe}(l), \text{it}_p) = \mathcal{E}(\text{Seq}(l, \text{it}_p))$</td>
<td>$+\sum_{t=m}^{n} \max_{i \in [1..m]} {4c_v + c_{f_m}, 5c_v + c_{f_i}}$</td>
</tr>
<tr>
<td></td>
<td>$+n(c_v + c_b) + \sum_{t=n+1}^{m-1} \max_{i \in [t-n+1..m]} {4c_v + c_{f_m}, 5c_v + c_{f_i}}$</td>
</tr>
</tbody>
</table>

Table 7.10: Summary of the computational costs per primitive

In the table we have assumed that $m$ is the number of stages in the list $l$; $\text{it}_s$ and $\text{it}_p$ are a sequential iterator and a parallel iterator, respectively, on the item list $\{x_1 \ldots, x_n\}$, $c_f$ is cost of a sequential function $f$ and $c_C$ is the cost of a generic inner primitive.
Chapter 8

Optimizing transformations

In this chapter we will explain how the cost system integrated into the semantic framework can be exploited to optimize the evaluation of the graph application. We will focus on the capability of our inference rules system to reduce an application graph into another functionally equivalent application graph. On the other hand, we will count on the computational cost system able to establish which of the two graphs exploits the minimum computational costs. In this scenario, once a functional equivalence is known or can be proved, the graph exploiting the minimum computational cost can be chosen for being executed.

8.1 Transformation analysis

As mentioned in chapter 3, the semantic framework designed in the previous chapters has the goal to provide an unambiguous description of a parallel programming model in which data parallel and control parallel concerns are kept orthogonal and independent. Moreover, the other great challenge is to improve the overall performance of the user application.

The semantic framework and the related computational cost system have been designed with this goal in mind. Indeed, they provide a semantic method to rewrite the application graph of primitives $G$ defined by the user as a composition of primitives into another graph of primitives $G'$ automatically individuated by the framework that exploits a better performance, i.e. a lower computational cost.

The main idea underlying our optimization mechanisms is to rewrite the graph $G$ provided by the user into a graph $G'$ defined by the framework if

$$\$(G, \text{it}) > \$(G', \text{it})$$

for some input iterator $\text{it}$.

The semantic framework defines $G'$ on the basis of two considerations. First of all, graph $G$ has to be rewritten into a functionally equivalent graph $G'$, where functionally equivalent means that $G$ and $G'$ are just two different control graphs expressing the producing the equivalent results on the same input dataset (see definition 3.4.1). We will prove such kind of equivalences by demonstrating that $G$ and $G'$ can be reduced to (or rewritten into) the same function, through the transformation rules provided by the semantic framework.

Thus, we will formally conclude that

$$G \leftrightarrow G'$$

Once the equivalence has been proved, it will be established that the user application could be indifferently executed by either evaluating $G$ or $G'$. At this point the computational cost system will help us to estimate which is the cheapest one and, consequently, which is the better candidate for the actual execution on the implementation framework.
We are going to present a set of well known equivalences between parallel constructs already documented in the literature. Our goal is to demonstrate that our semantic framework is able to cope with these well-known transformations. Moreover, the computational system we have introduced will allow us to decide whether an application graph has to be rewritten or not into its equivalent counterpart, in order to increase the performance application.

In the following, for each couple of graph \((G, G')\) we will discuss

- the demonstration of the functional equivalence between \(G\) and \(G'\);

- a computational costs analysis assessing which is the cheapest graph between \(G\) and \(G'\) in terms of computational costs and on which conditions.

Notational notes The relation \(\rightarrow^n\) labeled by \(n\) (or any \(\ast\)) means “after a certain number of transformations or steps”.

8.1.1 Apply(seqdo) vs seqdo(Apply)

Transformation 8.1.1 Let \(it: \text{SeqIterator}\) be an iterator on the item list \(\{x_1, \ldots, x_n\}\).

Let \(f = (f_1; \ldots; f_m)\) be the sequential composition of \(m\) sequential functions.

Let \(l' = [\text{Apply}(\text{Fun}(f_1)); \ldots; \text{Apply}(\text{Fun}(f_m))]\) be a list of \text{Apply} primitives each encapsulating one of the \(m\) sequential functions. Then, the following inequality holds

\[
$\text{Apply}(\text{Fun}(f)), it) \leq $\text{Apply}(\text{Fun}(f_1)); \ldots; \text{Apply}(\text{Fun}(f_m)), it) \tag{8.1}
\]

Let us rename \(G = \text{Apply}(\text{Fun}(f))\) and \(G' = (\text{Apply}(\text{Fun}(f_1)); \ldots; \text{Apply}(\text{Fun}(f_m)))\).

Functional equivalence As mentioned above, in order to compare the computational costs of \(G\) and \(G'\), we have to demonstrate that they are functionally equivalence, first. We will provide such demonstration by showing \(G \leftrightarrow G'\).

Starting from the definition of \text{Apply}:

\[
\mathcal{E}(\text{Apply}(f), it) \\
\rightarrow \{\text{by applying rule 6.23}\} \\
\mathcal{E}(f, f.B(\text{current}(it))) \parallel \mathcal{E}(\text{Apply}(f), \text{skip}(it)) \\
\rightarrow^n \{\text{by relaxing temporal constraints and letting } \forall i \in [1, n], it_i = f.B(\text{current}(i)(it))\} \\
\mathcal{E}(f, it_1) \parallel \ldots \parallel \mathcal{E}(f, it_n) \\
\rightarrow^\ast \{\text{by evaluating each argument of \text{pardo}}\} \\
\mathcal{E}(f_m(\ldots f_1(x_1)) \parallel \ldots \parallel f_m(\ldots f_1(x_n))\ldots) \\
\leftrightarrow \{\text{let } it_m \text{ be a sequential iterator on } \{f_{m-1}(\ldots f_1(x_1)\ldots), \ldots, f_{m-1}(\ldots f_1(x_n)\ldots)\}\} \\
\mathcal{E}(\text{Apply}(f_m), it_m) \\
\leftarrow \{\text{let } it_{m-1} \text{ a sequential iterator on } \{f_{m-2}(\ldots f_1(x_1)\ldots), \ldots, f_{m-2}(\ldots f_1(x_n)\ldots)\}\} \\
\mathcal{E}(\text{Apply}(f_{m-1}); \text{Apply}(f_m), it_{m-1}) \\
\leftrightarrow^\ast \{\text{rename } it_1 \text{ as } it\} \\
\mathcal{E}(\text{Apply}(f_1); \ldots; \text{Apply}(f_m)), it_1)
\]

The chain of transformations above has demonstrated that by relaxing the temporal constraints introduced by rule 6.23 (temporal constraints are not relevant for functional equivalence) and looking at the global evaluation process of \(G\), the final function evaluated by \(G\) is the same of \(G'\).
8.1. TRANSFORMATION ANALYSIS

Cost comparison From table 7.10 we know that

\[
G = (\text{Apply} (\text{Fun}(f_1; \ldots ; f_m)), it) = c_v \times (n + 1) + c_b \times n + \sum_{t=1}^{n} c_f
\]

where \(c_f = \sum_{i=1}^{m} c_{fi} \).

On the other hand, the sequential composition of \(m\) \textit{Apply}s costs

\[
G' = ((\text{Apply}(f_1); \ldots ; \text{Apply}(f_m)), it) = \sum_{t=1}^{m} c_{\text{Apply}(t)}
\]

\[
= \sum_{t=1}^{m} c_v \times (n + 1) + c_b \times n + \sum_{i=1}^{n} c_{fi}
\]

\[
= \sum_{t=1}^{m} c_v \times (n + 1) + c_b \times n + n \times c_f
\]

\[
= m \times c_v \times (n + 1) + m \times c_b \times n + n \times \sum_{t=1}^{m} c_{fi}
\]

\[
= m \times c_v \times (n + 1) + m \times c_b \times n + n \times c_f
\]

By comparing \(G\) and \(G'\) we have

\[
c_v \times (n + 1) + c_b \times n + n \times c_f \leq m \times c_v \times (n + 1) + m \times c_b \times n + n \times c_f
\]

meaning that

\[
G, it \leq G', it
\]

because each addendum of the left-side member of the inequality is lesser or equal to the corresponding right-side one. Moreover, the right side expression demonstrates that the actual difference in evaluating \(m\) \textit{Apply}s sequentially is given by the data transferring cost \(c_b\). In fact, in the second case, it has to be paid for each worker instantiation and for each \textit{Apply} instantiation. Conversely, in the first case \(c_b\) is paid only for transferring data to the remote workers.

Summarizing, \(\forall n, m > 0:\)

\[
(\text{Apply} (\text{Fun}(f_1; \ldots ; f_m)), it) \leq ((\text{Apply}(\text{Fun}(f_1)); \ldots ; \text{Apply}(\text{Fun}(f_m))), it)
\]
8.1.2 Apply(Pipe) vs Pipe(Apply)

Transformation 8.1.2 Let \( it_s : SeqIterator \) and \( it_p : ParIterator \), iterators on the same input data set \( \{x_1, \ldots, x_n\} \) and let \( f_1, \ldots, f_m \) a set of functions. Then, the following inequality holds

\[
\mathcal{E}(\text{Pipe}(l), it_s) = \{ \text{by applying rule 6.25, hasNext(it_s) \rightarrow true} \}
\]

\[
\mathcal{E}(\text{Pipe}(l), \text{hd}(l).\mathcal{B}(\text{current}(it_s))) \parallel \mathcal{E}(\text{Pipe}(l), \text{skip}(it))
\]

\[
\mathcal{E}(\text{Pipe}(l), it_{1,1}) \parallel \mathcal{E}(\text{Pipe}(l), it_{1,2})
\]

\[
\mathcal{E}(\text{Pipe}(l^2), \text{hd}(l^2).\mathcal{B}(\text{Apply}(f_1), it_{1,1})) \parallel \mathcal{E}(\text{Pipe}(l^2), \text{hd}(l^2).\mathcal{B}(\text{Apply}(f_1), it_{1,2}))
\]

\[
\mathcal{E}(\text{Pipe}(l^2), \text{hd}(l^2).\mathcal{B}(\text{Apply}(f_1), it_{1,m}))
\]

\[
\mathcal{E}(\text{Pipe}(l^2), \text{hd}(l^2).\mathcal{B}(\text{Apply}(f_1), it_{1,m}))
\]

\[
\mathcal{E}(\text{Pipe}(l^2), \text{hd}(l^2).\mathcal{B}(\text{Apply}(f_1), it_{1,1}))
\]

\[
\mathcal{E}(\text{Pipe}(l^2), \text{hd}(l^2).\mathcal{B}(\text{Apply}(f_1), it_{1,2}))
\]

\[
\mathcal{E}(\text{Apply}(\text{Pipe}(l'), it_p))
\]

For demonstrating that \( G \rightarrow G' \), we have relaxed the temporal constraint introduced by the evaluation of Pipe. On the other hand, in order to demonstrate the opposite reduction, thus \( G' \leftarrow G' \) we have added some temporal constraints that are not included in an Apply evaluation coupled with a parallel iterator, just to “mime” the pipeline temporal behavior. In both the cases such renounce/adding doesn’t harm the correctness of the functional equivalence.

Cost comparison Let us determine the cost of \( G = \text{Pipe}([\text{Apply}(f_1); \ldots; \text{Apply}(f_m)]) \). Since \( G \) is a pipeline application and we want to demonstrate inequality 8.2, let us suppose to have a balanced pipeline, i.e. a pipeline in which all the stages have the same computational cost \( c_f \).

Thus, let us considering to have a constant cost for each stage represented by an Apply application coupled with a parallel iterator. By applying rule 7.24 and by assuming that \( c_f \) is the cost of each sequential function \( f_i, i \in [1, m] \) we have:

\[c_f\]
$G = \$(\text{Pipe}(l, \text{it}_s)) = 3c_v + (m-1)(c_b + 6c_v) + (m-1)(3c_v + c_b + c_f) + \\
(n-m+1)(c_v + c_b) + (n-m+1)(8c_v + c_b + c_f) + \\
(m-1)(c_v + c_b) + (m-1)(5c_v + c_f) \\
= n(9c_v + 2c_b + c_f) + m(6c_v + c_b + c_f) - (3c_v + c_b + c_f)$

Let us determine the cost of $G' = (\text{Apply(\text{Pipe}([\text{Fun}(f_1); \ldots; \text{Fun}(f_m)]))}).$

From Tab.7.10 we can estimate

$\$(\text{Apply(\text{Pipe}([l_1; \ldots; l_m]), \text{it}_p)) = 3c_v + \max_{i \in [1, n]} c_b + \max\{c_{pipe}\}$

$= 3c_v + c_b + c_{pipe}$

The addendum $c_{pipe}$ refers to the pipeline each $\text{Apply}$’s worker instantiate. Such a pipeline applies the composition of the $m$ sequential functions starting from a singleton $x_i$. Thus, $\forall i \in [1, n]$, $c_{pipe}$ is equal to:

$E(\text{Pipe}(l(1)), \text{it}_{1,i}) \rightarrow 5c_v + c_{f_1}$
$\rightarrow c_v + c_b$
$E(\text{Pipe}(l(2)), \text{hd}(l(2)).B(y_{1,i}))$
$E(\text{Pipe}(l(2)), \text{it}_{2,i})$
$\rightarrow (m-2)(6c_v + c_b) + \sum_{k=1}^{m-2} c_{f_k}$
$\rightarrow 4c_v + c_{f_{m-1}}, y_i$
$\ldots$

and so on. Summarizing,

$c_{pipe} = \sum_{k=1}^{m-1} 6c_v + c_b + c_{f_k} + 4c_v + c_{f_m}$

In the meanwhile we have to take into account the hypothesis on the pipeline stages, all having the same computational cost $c_f$. Thus, the preceding cost $c_{pipe}$ becomes

$c_{pipe} = m(6c_v + c_b + c_f) - (c_b + 2c_v)$

Thus $\$(G') is equal to:

$\$(G') = m(6c_v + c_b + c_f) + c_v$

By comparing $\$(G' with $\$(G we have $\$(G) > $\$(G') if and only if

$n > \frac{2c_v + c_b + c_f}{9c_v + 2c_b + c_f}$

Let us call $Q = \frac{2c_v + c_b + c_f}{9c_v + 2c_b + c_f}$. As known, $c_f, c_f, c_b > 0$ since they are computational costs. Moreover, $2c_v + c_b + c_f < 9c_v + 2c_b + c_f$ hence, as a consequence, $0 < Q < 1$. In the meanwhile, $n > 1$ by definition of input data set, thus we can conclude that 8.2 always holds.
8.1.3 Apply(Pipe) vs Apply(seqdo)

Transformation 8.1.3 Let \( \text{it}_p : \text{ParIterator} \) be a parallel iterator on the input data set \( \{x_1, \ldots, x_n\} \) and let \( l = \{\text{Fun}(f_1); \ldots; \text{Fun}(f_m)\} \) a list of primitives of type Fun each encapsulating a sequential function. Then,

\[
\$\left(\text{Apply}(\text{Pipe}(l), \text{it}_p)\right) > \$\left(\text{Apply}((\text{Fun}(f_1); \ldots; \text{Fun}(f_m)), \text{it}_p)\right) \quad (8.3)
\]

Functional equivalence Starting from the definition of Apply

\[
\mathcal{E}(\text{Apply}(\text{Pipe}(l)), \text{it}_p) \rightarrow \{ \text{let } \forall i \in [1, \text{size}(\text{it}_p)] \text{Pipe.B}(\text{current}(\text{it}_p) : i) \rightarrow \text{it}_i \}
\]

\[
\equiv \mathcal{E}(\text{Pipe}(l), \text{it}_1) \parallel \cdots \parallel \mathcal{E}(\text{Pipe}(l), \text{it}_n)
\]

\[
\rightarrow \{ \text{(size(it}_i) + m - 2) \text{ steps later} \}
\]

\[
\mathcal{E}(\text{Fun}(f_m), \text{it}_{m-1}) \parallel \cdots \parallel \mathcal{E}(\text{Fun}(f_m), \text{it}_{m-1,n})
\]

\[
\rightarrow \{ \text{by evaluating } f_m \}
\]

\[
f_m(\ldots f_1(x_1) \ldots) \parallel f_m(\ldots f_1(x_2) \ldots) \parallel \cdots \parallel f_m(\ldots f_1(x_n) \ldots)
\]

\[
\leftarrow \{ \text{by definition of composition} \}
\]

\[
\mathcal{E}((\text{Fun}(f_1); \ldots; \text{Fun}(f_m)), \text{it}_1) \parallel \cdots \parallel \mathcal{E}(\text{Fun}(f_m); \ldots; \text{Fun}(f_1)), \text{it}_n
\]

\[
\equiv \left\{ \mathcal{E}(\text{Fun}(f_1); \ldots; \text{Fun}(f_m), \text{it}_i) \right\}_{i \in [1, n]}
\]

\[
\mathcal{E}(\text{Apply}((\text{Fun}(f_1); \ldots; \text{Fun}(f_m))), \text{it}_p)
\]

Cost comparison Following the cost estimation of Apply in table 7.10 concerning a parallel iterator as input, the cost of \( G' = \text{Apply}(\text{Fun}(f_1); \ldots; \text{Fun}(f_m)) \) is equal to

\[
\$\left(\text{Apply}((\text{Fun}(f_1); \ldots; \text{Fun}(f_m)), \text{it}_p)\right) = 3c_v + c_b + \sum_{i=1}^{m} c_{f_i}
\]

On the other hand, the cost of \( G = \text{Apply}(\text{Pipe}(l)) \) is equal to:

\[
\$\left(\text{Apply}(\text{Pipe}(l)), \text{it}_p\right) = 3c_v + c_b + c_{\text{pipe}}
\]

where, as in section 8.1.2,

\[
c_{\text{pipe}} = \left(\sum_{k=1}^{m-1} 6c_v + c_b + c_{f_k}\right) + 4c_v + c_{f_m} = 3c_v(2m - 2) + c_b(m - 1) + \sum_{i \in [1, m]} c_{f_i}
\]

Since \( c_{\text{pipe}} > \sum_{i=1}^{m} c_{f_i} > 0 \) it is quite clear that

\[
\$\left(\text{Apply}((f_1; \ldots; f_m)), \text{it}_p\right) < \$\left(\text{Apply}(\text{Pipe}(l)), \text{it}_p\right)
\]

if and only if

\[
(m - 1)(6c_v + c_b) > 0
\]

assuming that \( m > 1 \).
8.1.4 Pipe(Apply) vs seqdo(Apply)

Transformation 8.1.4 Let \( i_{t_s} \in \text{SeqIterator} \) and \( i_{t_p} \in \text{ParIterator} \) be two iterators of different type on the same input data set \( \{x_1; \ldots; x_n\} \). Let
\[
G = \text{Pipe}([\text{Apply}(\text{Fun}(f_1)); \ldots; \text{Apply}(\text{Fun}(f_m))])
\]
such that \( \text{Apply} \mathcal{B}(x) \in \text{ParIterator} \) for each instantiation of \( \text{Apply} \). Moreover, let
\[
G' = (\text{Apply}(\text{Fun}(f_1)); \ldots; \text{Apply}(\text{Fun}(f_m)))
\]
Then,
\[
\$G, i_{t_p}$ \geq \$G', i_{t_p}$ \tag{8.4}
\]

Functional equivalence By starting the evaluation of \( G \) from the definition of \( \text{Pipe} \)
\[
\mathcal{E}(\text{Pipe}([\text{Apply}(\text{Fun}(f_1)); \ldots; \text{Apply}(\text{Fun}(f_m))]), i_{t_p})
\]
\[
\rightarrow \{\text{let } \text{Apply}(\text{Fun}(f_1)); \mathcal{B}(x_1) \rightarrow i_{t_1,1}\}
\]
\[
\mathcal{E}(\text{Pipe}(l), i_{t_1,1}) \parallel \mathcal{E}(\text{Pipe}(l), i_{t(2)})
\]
\[
\rightarrow \{\text{by renouncing to temporal constraints}\}
\]
\[
\mathcal{E}(\text{Pipe}(l'), \text{hd}(l'), \mathcal{B}(\text{Apply}(\text{Fun}(f_1)), i_{t_1,1})) \parallel []
\]
\[
\rightarrow \{\mathcal{E}(\text{Apply}(\text{Fun}(f_1)), i_{t_1}) \rightarrow y_1 \land \text{hd}(l'). \mathcal{B}(y_1) \rightarrow i_{t_2}\}
\]
\[
\mathcal{E}(\text{Pipe}(l'), i_{t_2})
\]
\[
\rightarrow \mathcal{E}(\text{Pipe}(l''), i_{t_m})
\]
\[
\rightarrow \mathcal{E}(\text{hd}(l''), i_{t_m})
\]
\[
\rightarrow \mathcal{E}(\text{Apply}(\text{Fun}(f_1)), i_{t_m})
\]
\[
\leftarrow^* \mathcal{E}(\text{Apply}(\text{Fun}(f_2)); \ldots; \text{Apply}(\text{Fun}(f_m))), i_{t_2})
\]
\[
\leftarrow \{\text{Apply}(\text{Fun}(f_1)); \ldots; \text{Apply}(\text{Fun}(f_m))), i_{t_2}) \rightarrow i_{t_1} \land
\]
\[
\mathcal{E}(\text{Apply}(\text{Fun}(f_1)), i_{t_1}) \rightarrow y_1 \land
\]
\[
\text{Apply}(f_2). \mathcal{B}(y_1) \rightarrow i_{t_2}\}
\]
\[
\mathcal{E}(\text{(Apply}(\text{Fun}(f_1)); \ldots; \text{Apply}(\text{Fun}(f_m))]), i_{t_p})
\]

Cost comparison Let \( \forall i \in [1, n]. \mathcal{B}(x_i) \rightarrow c_x \) it meaning that building an iterator on a singleton of the input data set \( x_i \) costs \( c_x \). Since the cost of building an iterator is proportional to the dimension of the dataset, let \( n \times c_x \) be the cost of building the parallel iterator upon the whole input data set.

Since we for each \( \text{Apply} \) instantiation the respective builder creates a parallel iterator on the whole data set, we can easily say that the cost of the right-side member is equal to
\[
\$G', i_{t_p} = \$((\text{Apply}(\text{Fun}(f_1)); \ldots; \text{Apply}(\text{Fun}(f_m))), i_{t_p}) = \sum_{i=1}^{m} 3c_w + n \times c_x + c_{f_i}
\]

In order to estimate the cost of the pipeline, we have to notice that its input iterator is a parallel iterator. This means that all the elements are accessed as a whole and on such set of input elements, the first \( \text{Apply} \) will built is own iterator (a parallel iterator for our hypothesis). Then, the transformation chain of the pipeline will follow the case in which the number of stages is greater or equal to one but the iterator size is just 1. Under this condition, as we have seen in rule 8.1.2
the cost of a pipeline is given only by such a transformation chain. The cost of the outer pipeline is equal to (let $c_{\alpha}$ be the cost of $\text{Apply}(f_i)$):

$$\$(G, it_p) = \sum_{k=1}^{m-1} (6c_v + n \times c_x + c_{\alpha_k}) + 4c_v + n \times c_x + c_{\alpha_m}$$

The cost of each $\text{Apply}$ instantiation is equal to

$$c_{\alpha_i} = 3c_v + c_x + c_{f_i}$$

Summarizing

$$\$(G, it_p) = \sum_{k=1}^{m-1} (6c_v + n \times c_x + c_{\alpha_k}) + 4c_v + n \times c_x + c_{\alpha_m}$$

$$= \sum_{k=1}^{m-1} (6c_v + n \times c_x + 3c_v + c_x + c_{f_i}) + 4c_v + n \times c_x + 3c_v + c_x + c_{f_m}$$

$$= (m - 1) \times (9c_v + c_x \times (n + 1)) + \left(\sum_{i=1}^{m} c_{f_i}\right) + 7c_v + n \times c_x$$

$$= 9mc_x + mc_x(n + 1) + 7c_v + \sum_{k=1}^{m} c_{f_i}$$

By comparing the two costs we have $\$(G, it_p) \geq \$(G', it_p)$, since

$$9mc_x + mc_x(n + 1) + 7c_v + \sum_{k=1}^{m} c_{f_i} \geq m \times (3c_v + nc_x) + \sum_{k=1}^{m} c_{f_i}$$

In fact, the previous equation holds if

$$m \geq \frac{9c_x - 7c_v}{10c_x - 3c_v}$$

Moreover, we have the implicit constraint $m \geq 1$, i.e. it have to be

$$\frac{9c_x - 7c_v}{10c_x - 3c_v} \geq 1$$

must hold, that is

$$-4c_v > c_x$$

But this is always true since $c_v$ and $c_x$ are positive values. Then 8.5 always holds.
8.1.5 Normal Form

Transformation 8.1.5 Let \( l = [\text{Fun}(f_1); \text{Apply}((\text{Fun}(g)); \text{Fun}(f_2))] \) and let \( it_s \in \text{SeqIterator} \) as well as \( it_p \in \text{ParIterator} \) two iterators on the same input data set \( D = \{x_1, \ldots, x_n\} \). Then,

\[
\$(\text{Pipe}(l), it_s) < \$(\text{Apply}((\text{Fun}(f_1; g; f_2)), it_s) \quad (8.6)
\]

The demonstration proceeds in three steps:

1. first of all we demonstrate that given \( x \in D \) and built \( it \) onto \( x \) such that \( \text{size}(it) = 1 \),

\[
\mathcal{E}(\text{Apply}(C, it)) \equiv \mathcal{E}(C, it)
\]

2. second, we demonstrate that a pipeline is functionally equivalent to the sequential composition of its stages. In particular, given an iterator \( it \) such that \( \text{size}(it) = 1 \)

\[
\mathcal{E}(\text{Pipe}(\text{Fun}(f_1); \text{Fun}(g); \text{Fun}(f_2)), it) \equiv \mathcal{E}(\text{Fun}(f_1; g; f_2), it)
\]

3. third we can conclude that given an iterator \( it \) such that \( \text{size}(it) = 1 \)

\[
\mathcal{E}(\text{Pipe}([\text{Fun}(f_1); \text{Apply}(g); \text{Fun}(f_2)]), it)
\]

\[\equiv \text{by applying demo 1} \]

\[
\mathcal{E}(\text{Pipe}(\text{Fun}(f_1); \text{Fun}(g); \text{Fun}(f_2)), it)
\]

\[\equiv \text{by applying demo 2} \]

\[
\mathcal{E}(\text{Fun}(f_1; g; f_2), it)
\]

4. we demonstrate that by extending the preceding result to all the input data set, an \text{Apply} primitive is adopted, thus

\[
\mathcal{E}(\text{Pipe}..., it_s) \equiv \mathcal{E}(\text{Apply}(..), it_s)
\]

Step 1 We want to demonstrate that given an iterator \( it \) such that \( \text{size}(it) = 1 \), then

\[
\mathcal{E}(\text{Apply}(C), it) \equiv \mathcal{E}(C, it)
\]

The equivalence is very easy to demonstrate by analyzing the evaluation process of \text{Apply}. Let us consider the case \( it \in \text{SeqIterator} \)

\[
\mathcal{E}(\text{Apply}(C, it))
\]

\[\rightarrow \{ \text{it} \in \text{SeqIterator} \text{ and by applying rule 6.23} \}
\]

\[
\mathcal{E}(C, C.B(\text{current}(it))) || \mathcal{E}(\text{Apply}(C), \text{skip}(it))
\]

\[\rightarrow \{ \text{by reducing } C.B(\text{current}(it)) \xrightarrow{3} \text{it}' \land \text{skip}(it) \xrightarrow{5} \text{it}(2) \}
\]

\[
\mathcal{E}(C, \text{it}') || \mathcal{E}(\text{Apply}(C), \text{it}(2))
\]

\[\rightarrow \{ \text{by applying rule 6.17 and 6.24} \}
\]

\[
\mathcal{E}(C, \text{it}')
\]

The evaluation of \text{Apply} reduces to a \text{pardo} expression whose second argument reduces to the empty set \([\text{it}']\), since \text{hasNext}(\text{skip}(\text{it})) \rightarrow \text{false}. Instead, the first argument represents the evaluation of \( C \), to which the expected iterator is given as argument. Nevertheless, \( \text{it}' \) carries the same value as \( \text{it} \) and, since they are both iterator with size 1, their type don’t influence the behavior.
of their related primitives. Thus, from a functional point of view, evaluating \( \text{Apply}(C) \) or \( C \) directly is the same matter, because the first leads to the evaluation of the second in only one step that doesn’t change the semantics of the evaluation.

Now, let us consider \( it \in \text{ParIterator} \). Also in this case, since \( \text{size}(it) = 1 \) we have the transformation and the same considerations given above hold also in this case.

Summarizing, whenever we have an iterator \( it \) such that \( \text{size}(it) = 1 \), \( \mathcal{E}(\text{Apply}(C), it) \) can be substituted by \( \mathcal{E}(C, it) \).

**Step 2** By evaluating \( \text{Pipe} \), we get

\[
\begin{align*}
\mathcal{E}(& \text{Pipe}([\text{Fun}(f_1); \text{Apply}(\text{Fun}(g)); \text{Fun}(f_2)], it_s) \\
& \quad \rightarrow \{ \text{let } \text{Apply}(\text{Fun}(f_1)).B(current(it_s)) \rightarrow it_{1_s} \land \text{skip}(it_s) \rightarrow it_s^{(2)} \}
\end{align*}
\]

As it can be seen looking at the definition of \( it_{1,n} \), such iterator encapsulates only one task (i.e. \( \text{current}(it_s) = x_i \)). Moreover, this type of step occurs for each element provided by the iterator, that is \( \forall j \in [1,n].\text{size}(it_{s,j}) = 1 \). This means each time the evaluation of a task \( x_j \) is started by reducing the expression \( \mathcal{E}(\text{Pipe}(l), it_{1,j}) \), rule 6.27 will be applied, that is the rule concerning a list of stages longer than 1 and an iterator size equal to 1. This rule sequentially applies the first primitive listed in \( l \) to the iterator given as input; it builds a new iterator on the result obtained by the first application and passes such iterator to the second primitive listed in \( l \) and so on, till the final result. This process corresponds to a sequential application of a list of primitive to a given starting value.

Thus, given an iterator \( it \) such that \( \text{size}(it) = 1 \) we can strongly conclude that

\[
\mathcal{E}(\text{Pipe}([C_1; \ldots; C_m], it)) = \mathcal{E}(\text{Fun}([C_1; \ldots; C_m], it))
\]

**Step 3** From step 1 and step 2 we have deduced that given an iterator \( it \) such that \( \text{size}(it) = 1 \),

\[
\mathcal{E}(\text{Pipe}([\text{Fun}(f_1); \text{Apply}(\text{Fun}(g)); \text{Fun}(f_2)], it) = \mathcal{E}(\text{Fun}(f_1; g; f_2), it)
\]

What happens if, instead of \( it \) as input iterator, we have \( it_s \) such that \( \text{size}(it_s) > 1 \)? As we can see from applying rule 6.25, task \( x_i \) starts the evaluation process through the expression \( \mathcal{E}(\text{Pipe}(l), \text{hd}(l).B(x_i)) \). This evaluation involves an iterator of size 1, thus all the consideration given above fix to this case. Furthermore, we have

\[
\begin{align*}
\mathcal{E}(& \text{Pipe}(l), it_s) \\
& \quad \rightarrow \{ \text{by applying rule 6.25} \} \\
& \quad \rightarrow \{ \text{since } \text{hd}(l).B(x_1) \rightarrow it_{1,1} \} \\
& \quad \rightarrow \mathcal{E}(\text{Pipe}([\text{Fun}(f_1); \text{Apply}(\text{Fun}(g)); \text{Fun}(f_2)], it_{1,1}) \| \mathcal{E}(\text{Pipe}(l), it_s^{(2)})
\end{align*}
\]

At this point we have demonstrated that the left argument of the equivalence can be substituted by \( \mathcal{E}(\text{Fun}(f_1; g; f_2), it_s) \). Moreover, the right argument is a recursive call to \( \mathcal{E} \), thus the same reasoning can be applied to the next evaluation step and so on, until the iterator \( it_s \) will provide tasks. Thus, we can rewrite
\[ E(\text{Pipe}(l), it_s) \]
\[ \rightarrow \{ \text{by applying rule 6.25} \} \]
\[ E(\text{Pipe}(l), hd(l).B(x_1)) \parallel E(\text{Pipe}(l), \text{skip}(it_s)) \]
\[ \rightarrow \{ \text{since } hd(l).B(x_1) \rightarrow it_{1,1} \} \]
\[ E(\text{Pipe}(\text{Fun}(f_1); \text{Apply}(\text{Fun}(g)); \text{Fun}(f_2)), it_{1,1})) \parallel E(\text{Pipe}(l), it_s^{(2)}) \]
\[ \rightarrow \{ \text{by applying step 1 and 2 of the demonstration} \} \]
\[ E(\text{Fun}(f_1; g; f_2), it_{1,1}) \parallel E(\text{Fun}(f_1; g; f_2), it_s^{(2)}) \]

But the last evaluation step is just the one related to an \text{Apply} primitive getting the sequential iterator \( it_s \) as input. In fact, the following steps can be done backwards

\[ \cdots \]
\[ E(\text{Fun}(f_1; g; f_2), it_{1,1}) \parallel E(\text{Fun}(f_1; g; f_2), it_s^{(2)}) \]
\[ \leftarrow \{ \text{let } it_{1,1} = \text{Fun}(f_1; g; f_2).B(\text{current}(it_s)) \} \]
\[ E(\text{Fun}(f_1; g; f_2), \text{Fun}(f_1; g; f_2).B(\text{current}(it_s))) \parallel \text{Apply}(\text{Fun}(f_1; g; f_2), \text{skip}(it_s)) \]
\[ \leftarrow \{ \text{by rule 6.23} \} \]
\[ E(\text{Apply}(\text{Fun}(f_1; g; f_2)), it_s) \]

This means that the same pattern of computation provided by the evaluation of \( E(\text{Pipe}(l), it_s) \), can be also provided by \( E(\text{Apply}(\text{Fun}(f_1; g; f_2)); it_s) \)

**Cost comparison** Let us suppose that \( \text{size}(it_s) = n, \$ (f_1) = c_{f_1}, \$ (f_2) = c_{f_2}, \$ (g) = c_g \) and \( c_{\text{fun}} = \$ (\text{Fun}(f_1; g; f_2)) = c_{f_1} + c_{f_2} + c_g \).

**Cost of \text{Pipe}** Let us consider the computational cost for evaluating \( \text{Pipe}(l) \) providing \( it_s \) as input iterator. The pipeline reaches its best performance when its stages are completely balanced, i.e. when they require a uniform computational cost. In our case this means that \( c_{f_1} = c_{f_2} = c_{\text{Apply}} \).

Then, considering that \( m = 3 \) and that the costs of the stages are all equal to \( c_{f_1} \) the cost of a \text{Pipe} primitive is given by Tab.7.10, provided that \( c_v \) is the cost of a local function evaluation and \( c_b \) is the cost of building an iterator. Thus,

\[ \$ (\text{Pipe}(l), it_s) = 3c_v + 2 \times (c_b + 6c_v) + \sum_{t=1}^{2} \max \{ c_f \} + (n - 1) \times (c_v + c_b) + \sum_{t=3}^{5} \max_{i \in \{1, m\}} \{ 4c_v + c_f, 5c_v + c_{f_1}, 5c_v + c_g \} + 2 \times (c_v + c_b) + \max \{ 4c_v + c_f, 5c_v + c_g \} + 4c_v + c_{f_2} \]

\[ = 3c_v + 2 \times (c_b + 6c_v) + 2c_{f_1} + (n - 1)(c_v + c_b) + (5c_v + c_{f_1}) \times (n - 3 + 1) + 2(c_v + c_b) + 5c_v + c_{f_1} + 4c_v + c_{f_1} \]

and by reducing the addendum we have

\[ \$ (\text{Pipe}(l), it_s) = n(6c_v + c_b + c_{f_1}) + 15c_v + 3c_b + 2c_{f_1} \] \hspace{1cm} (8.7)

**Cost of \text{Apply}** Looking at Tab.7.10, we know that in case of providing a sequential iterator as argument, an \text{Apply} primitive requires the following computational cost:
\[ $(\text{Apply}(\text{Fun}(f_1; g; f_2)), \text{it}_s) = \quad c_v \times (n + 1) + c_b \times n + \sum_{t=1}^{n} c_{\text{fun}} \]
\[ = \quad c_v \times (n + 1) + c_b \times n + c_{\text{fun}} \times n \]
\[ = \quad n(c_v + c_b + c_{\text{fun}}) + c_v \]

But we are in the particular case in which \( c_{f_1} = c_{\text{Apply}} \). \( c_{\text{Apply}} \) is the cost of the inner \( \text{Apply} \) primitive instantiated by the pipeline. As we have seen above, such primitive receives as argument a sequential iterator of size 1, thus
\[ c_{\text{Apply}} = 2c_v + c_b + c_g \]

but this also means that
\[ 2c_v + c_b + c_g = c_{f_1} \Rightarrow c_g = c_{f_1} - 2c_v - c_b \]

Consequently, the cost of the \( \text{Apply} \) primitive in case of balanced stages of the pipeline is equal to
\[ $(\text{Apply}(\text{Fun}(f_1; g; f_2)), \text{it}_s) = c_v(n + 1) + nc_b + (3c_{f_1} - 2c_v - c_b) \] (8.8)

**Final comparison** We want now to demonstrate that the inequality
\[ $(\text{Pipe}([\text{Fun}(f_1); \text{Apply}(g); \text{Fun}(f_2)]), \text{it}_s) > $(\text{Apply}(\text{Fun}(f_1; g; f_2)), \text{it}_s) \]
always holds.

In fact, by comparing the cost in (8.7) and the one in (8.8) we have
\[ n(6c_v + c_b + c_{f_1}) + 15c_v + 3c_b + 2c_{f_1} > c_v(n + 1) + nc_b + (3c_{f_1} - 2c_v - c_b) \]
if and only if
\[ c_{f_1} \geq \frac{-(16c_v + 4c_b + 5nc_v)}{n - 1} \] (8.9)

But it has to be considered that \( n > 1 \) because of our hypothesis on the size of \( \text{it}_s \); thus, provided that \( c_v, c_b \geq 0 \) since they represent computational costs that can’t be negative values, \( \forall n > 1 \) the following inequality is always true
\[ \frac{-(16c_v + 4c_b + 5nc_v)}{n - 1} < 0 \]

In the meanwhile, also \( c_{f_1} \) represents a computational cost, i.e. \( c_{f_1} > 0 \). Thus,
\[ \frac{-(16c_v + 4c_b + 5nc_v)}{n - 1} < 0 < c_{f_1} \]
and the inequality 8.9 is always true.

### 8.2 Remarks

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 [LAG01, ABH04] and, possibly, at associating a cost value to its execution [AD98].

In this chapter we have presented a new approach for the evaluation of the computational cost based on the analysis both of the structure of the application graph and on its parallel behavior. In particular, in chapter 6 we have given a set of inference rules describing the evaluation process of each primitive of ours. In this chapter we have extended the semantics expressed by the system of inference rules by labeling each represented transformation with its related cost. Such cost
depends on the primitive types involved in the transformation and on their parallel behavior can be influenced by the particular coupled iterator instance.

The sample application we have provided shows the feasibility and the characteristics of our approach. We are able to statically establish if two graphs are functionally equivalent. In this case, we are able to substitute the user application graph with another equivalent graph exploiting a better overall performance. Moreover, the cost system has evidenced that the Pipe primitive is an odd control structure if considering its computational cost. In fact, in all the cases provided, a (sub)graph in which a Pipe appears can always be improved by an Apply encapsulating a sequential composition of the stages. This result is not new. In fact, [DT02] and [ADT03] already obtained the same result through other ways, thus demonstrating that our approach is able to introduce new knowledge on the application concerns as well as coherent with the known literature.

The transformations considered in this thesis are not the only ones our framework can exploit. The set of transformation can be extended providing to the framework new know-how about graph optimization. Also the set of primitives can be extended, thus enriching the set of inference rules representing the evaluation process and providing new case study to the optimization mechanisms.
Chapter 9

A prototype implementation

Our semantic framework has been studied and developed in parallel with a prototype implementation aiming at testing the feasibility of our approach. Such implementation provided us a feedback about our theory on the usability of the model and, on the other hand, on the concrete advantages carried out by our abstractions (primarily, iterators and primitives) both in terms of orthogonality and in terms of optimization.

The result is a Java package providing a set of classes that the user can instantiate, extend, implement etc., in order to encapsulate its application into our programming model.

General structure  The main package `it.unipi.di.sonia` is composed of two inner packages:

- `it.unipi.di.sonia.adts` providing all the classes needed to implement the data abstractions. With respect to the semantic framework we have presented, the implementation framework provides classes for instantiating `ArrayView` and `MatrixView` objects. We have given higher priority to these views with respect to the others because they suffice to answer our need of understanding the feasibility of the programming model.

  The content of this package is depicted in Fig.9.1 where dot boxes indicate interfaces while solid boxes represent implementation classes.

- `it.unipi.di.sonia.collectives` providing all the classes needed to instantiate the primitives given by the framework. Fig. 9.2 shows the general content of this package.

System requirements  We have chosen Java as programming language for implementing the programming framework in order to take advantage of the numerous facilities provided by its platform. First of all, Java is based on a strong type checker that allows us to avoid several programming errors, thus saving developing time. Moreover, the object-oriented paradigm lead us to an easy encoding of the entities implementing the model: primitives as well as abstraction on data have been implemented thanks to the facilities provided by the object orientation paradigm. In fact, by exploiting them, the Java model allowed us to focus our attention on the representation of the semantic model rather than on other low level implementation details. As an example, by using RMI (Remote Method Invocation) classes as a remote procedure call system available on-the-shelf, we have been allowed to abstract from interprocess communication details while parallelizing the applications.

In the following we will detail the implementation framework from a user point of view. In particular, we will discuss how abstractions on data and abstractions on control can be used in order to build a parallel application. Thanks to the orthogonalization we have reached through our formalism, it will be possible to separately discuss the data concerns and control concerns: we will be able to program data accesses without referring to any directive on the control and, vice versa,
CHAPTER 9. A PROTOTYPE IMPLEMENTATION

Figure 9.1: it.unipi.di.sonia.adt package structure

Figure 9.2: it.unipi.di.sonia.collectives package structure
we will be able to instantiate a graph of primitives (the application itself), without specifying how data will be accessed.

The presentation of the implementation framework will be equipped with a number of examples aiming at simplifying the user interface. Moreover, the orthogonality will be evident in examples in which the same data abstractions will provide tasks for different graphs and, vice versa, the same graph can be reused to compute different applications on different data abstractions.

The presentation of the programming framework will begin with the description of data abstractions (abstract data types and view types), followed by the description of the iterator type (and related access patterns).

Consequently, the set of primitives will be presented as basic-building blocks for structuring a parallel application.

Eventually, the evaluation process will be analyzed by means of the class `Evaluator` implementing it.

## 9.1 Data abstractions and iterators

In this section we will show how input data and data accesses can be programmed into our programming framework.

The main goal is evidencing that our abstraction mechanisms are:

- **linguistically expressive**: i.e. they can describe data and access behaviors through a very user-friendly interface
- **extensible and flexible**: they cope with a great variety of data descriptions
- **able to actually describe access patterns on the input data without either linguistic or conceptual influences** coming from control parallel concerns that don’t need to be mixed or expressed at the same level.

In particular, the third point will be outlined by the fact that we will able to separately program data concerns and control primitives.

### 9.1.1 Abstract data types

Data abstractions and iterator types are provided within the package `it.unipi.di.sonia.adt` that provides all the classes needed to implement the data abstractions. In particular, `DataContent` is the interface for all those classes implementing *abstract data types*.

Formally, a `DataContent` object is described by the grammar

```
PrimitiveObject ::= Integer | Float | String | Object ....
DataContent ::= PrimitiveObject | list of DataContent
```

In fact, the minimal unit of computation is an object implementing a `DataContent`. Such object could be a *primitive object* (for example, an `Integer` object or an object defined by the user) or it could be a set of `DataContent` objects.

The user can implement a new abstract data type to be encapsulated into the framework

1. by simply implementing the interface `DataContent`, i.e. by providing a new implementation of the list representing in the grammar a `DataContent` set, or

2. by extending the provided implementations. The customization of the abstract data type can be done by overriding the method `init()` of a `DataContent` implementation, in order to initialize the data set as the programmer decides.
In both cases, the data set is just a set of elements indexed by a linear position. Such set has no structure or logical organization apart from those given by the linear, sequential position of the single elements and, as the DataContent interface will show, operations for getting, setting, removing and inserting elements are the only one provided.

```java
public interface DataContent {
    public void init();
    public int capacity();
    public int size();
    public DataContent elementAt(int pos);
    public void setElementAt(DataContent d, int pos);
    public boolean isPrimitive();
    public Object getValue();
    public void printContent();
    public ArrayView getArrayView();
    public MatrixView getMatrixView(int i, int j);
    public void append(DataContent d);
    public void add(DataContent content2);
    public void removeAt(int i);
}
```

A set of DataContent implementations is already given within the framework: DataContentSet and DataContentArray are two different implementations of the interface DataContent, implementing list of DataContent as a java.util.Vector and as an array, respectively.

**Example 9.1.1** An implementation of the DataContent interface, could be given by the user defined class DataContentFile in which the raw input data is represented by the content of an input file. The abstraction could shadow read and write operations on the file by, for instance, buffering lines or blocks of data. In any case the structure of the new class is the following

```java
public class DataContentFile implements DataContent {
    private String filename;
    private BufferedReader br;

    // class constructor definition
    public DataContentFile(String fname){
        filename = fname;
        init();
    }

    public void init(){
        //open the file in read mode
        br = new BufferedReader(new FileReader(filename));

        //buffer the content of the file
        while ((s = br.readLine()) != null )
            this.add(s);
        ...
    }
    // implementation of the requested method ...
}
```

while in the application main the user will create its abstract data type by instantiating an object of type DataContentFile, thus by simply writing

```java
DataContent dcf = new DataContentFile("filename");
```
Example 9.1.2 Let us suppose we want to instantiate a set of \( n \) integers randomly generated. In such case, we can inherit all the functionalities of class `DataContentSet` but we need to override the initialization method. Thus, in the main program the user can specialize class `DataContentSet` in line by writing

```java
// instantiate a DataContentSet of cardinality n
DataContent dataset = new DataContentSet(n){
    public void init(){
        Random r = new java.util.Random();
        for (int i = 0 ; i < n ; i++ )
            this.setElementAt(new Integer(r.nextInt()),i);
    }
}
```

9.1.2 View instantiation

As explained in chapters 3 and 4, a complex logical structure and organization can be given to a raw input data set by instantiating a view on it. A view is a sort of filter on the flat set of input tasks that allows to access them through pattern rules that don’t reflect on the actual implementation of the data set.

There are at least two advantages in structuring by means of views.

First of all, the same input data set can be thought as organized in different manner (and thus, it will be logically accessed through different rules) without requiring multiple implementations. Thus, a single implementation or instantiation suffices to represent the input data set, saving development time. On the other hand, different algorithms can be implemented on the same input data set by simply changing the views adopted. For instance, items can be gathered or distributed without altering their implementation and/or their location because their organization is just a logical matter.

Another advantage comes from the field of distributed programming. In fact, a forthcoming implementation of the `DataContent` interface, will provide the possibility to instantiate distributed data structures. By applying, for instance, an `ArrayView` on a distributed data structure, the programmer can write its code referring to a simple logical structure as the array one (thus, indexing elements positions) and without concerning any the details related to interprocess communication, caching mechanism etc., or, in other words, to the actual implementation of data.

In chapter 4 we have formalized five view types:

- **ArrayView**: each element of the input data set is considered as an element of an array structure, thus member of a set of contiguous elements. As such, it can be accessed by expressing its position into the structure. An array view \( v \) is created from a raw input data set \( s \) by applying an homomorphism between positions of \( s \) and positions of \( v \).

- **MatrixView**: each element of the input data set is considered as an element of a \( N \times M \) matrix structure, thus member of a set of elements indexed by a couple \((row\_index, col\_index)\). A matrix view \( m \) is created from a raw input data set \( s \) by applying an homomorphism between \( N \) contiguous positions of \( s \) and rows of \( m \).

- **ListView**: each element of the input data set is considered as an element of list structure, thus member of a set of elements provided with a relation of "predecessor" and/or "successor". A list view \( l \) is created from a raw input data set \( s \) by applying an homomorphism between positions of \( s \) and elements of \( l \).

- **GraphView**: each element of the input data set is considered as a vertex of a graph data structure, thus member of a set of elements provided with a relation of neighborhood. A graph view \( g \) is created from a raw input data set \( s \) by applying an homomorphism between elements of \( s \) and vertexes of \( g \), i.e. by instantiating neighborhood relations between vertexes of \( g \).
• **TreeView**: each element of the input data set is considered as a node of a tree data structure, thus member of a set of elements provided with a father-child relationship. A tree view \( t \) is created from a raw input data set \( s \) by applying an homomorphism between elements of \( s \) and nodes of \( t \), i.e. by instantiating descendant relations between nodes of \( t \).

Currently, also the first two views are given within the implementation framework, since we intended to start our experiments with structures often occurring in parallel applications and proving a logic on which various reasonings on partitioning and/or distribution strategies can be applied without much effort. The implementation of **ListView**, **GraphView** and **TreeView** is therefore an objectives of the future work.

A view can be instantiated by invoking its constructor that, in all the cases, requires as argument the abstract data type to which the view has to be applied. As an example, the constructor for a **MatrixView** object has the synopsis

```
public MatrixView(DataContent d, int rows, int cols)
```

while the constructor for the **ArrayView** has the following one

```
public ArrayView(DataContent d)
```

Moreover, each class implementing a certain abstract data type provides factory methods for instantiating different type of views.

This means that the object dataset defined in example 9.1.2 will provide at least two methods, as depicted from the **DataContent** interface:

```
public ArrayView getArrayView();
```

setting an array view on the input data set, and

```
public MatrixView getMatrixView(int rows, int cols);
```

setting a matrix view of dimensions \( \text{rows} \times \text{cols} \) on the input data set.

Obviously, both methods contain invocations to the constructor of the related view as the ones given above but the construction is safer than directly invoking the view constructor because the arguments (as, for instance, the number of rows and columns) are parsed and errors are avoided.

**Example 9.1.3** Once an input data set has been instantiated, we need to assign a structure to its elements. A **MatrixView** structure of \( r \) rows and \( c \) columns can be given by simply adding to the instructions in example 9.1.2 the last line:

```
//instantiate a DataContentSet of cardinality n
DataContent dataset = new DataContentSet(n){...

MatrixView mv = dataset.getMatrixView(r,c)
// or MatrixView mv = new MatrixView(dataset,r,c);
```

### 9.1.3 Patterns of access and iterators

Once a view has been assigned to an input data set, we have declared which global structural organization of data we will think about. In order to refine the access and to complete its description, we need two further steps:

1. we have to declare the minimal unit of access. Coherently with the given view, we have to establish which is the piece of view we will atomically access. In other words, once we have declared that, for instance, our input data set is a matrix of dimension \( r \times c \), we have to specify if we will access the matrix by punctually reading/writing \( r \times c \) elements, rows of elements, columns of elements, or rectangular blocks of elements.
2. we have to declare if each minimal unit of access composing the view will be accessed sequentially or in parallel. For instance, in the case of an \((r \times c)\)-matrix accessed by rows we have to declare if each row will be accessed one after the other or in parallel as a whole. In the first case, the iterator will return a row as a singleton of the view; in the latter case, the iterator will return a set of rows.

Step 1 is done by selecting an \texttt{AccessPattern} that characterizes the access, while step 2 is done by instantiating a \texttt{SeqIterator} object or a \texttt{ParIterator} object.

**Example 9.1.4** A matrix is a very flexible structure that can be accessed in different ways. The framework provides a set of \texttt{AccessPattern} implementations both for the matrix view and for the array view. Let us take into account the matrix view defined in example 9.1.3. Such matrix could be accessed by single (or multiples of) rows, meaning that the minimal unit of access is a block of \(r\) contiguous elements (or their multiples).

In case of a \texttt{GraphView} instead, by selecting an \texttt{AccessPattern}, one establishes how (and/or in which order) the vertexes of the graph (and its neighbors) have to be accessed. For example, the minimal access unit can be a single vertex plus its direct neighbors.

In a tree view, the access pattern \texttt{LevelOrderPattern} establishes that both the unit of access (for instance, a node of the tree corresponds to an element of the data set) and the order by which such unit will be provided with respect to the others in the view.

Since the iterator instance is strictly related to the logical structure on which it is built, the view class provides the factory methods for instantiating both sequential and parallel iterators. In fact, the \(V\) interface, contains the two factory methods

```java
public Iterator getParIterator(AccessPattern ap);
public Iterator getSeqIterator(AccessPattern ap);
```

instantiating a parallel iterator and a sequential iterator, respectively.

Coherently with the type of view, the factory methods accept as input an instance of \texttt{AccessPattern}.

In fact, the role of an iterator is furnishing (in parallel o sequentially) the units of access provided by the view and, as explained above, such units are represented by the \texttt{AccessPattern} selected by the user on this view.

In the following example we will show the \texttt{MatrixView} case but its model of usage can be easily extended to any other type of view.

**Example 9.1.5** Let us consider again the program in example 9.1.3. Let us suppose we are working on an image processing algorithm requiring that the \(r \times c\) matrix must be accessed by disjoint blocks of items. Let \(x \times y\) be the dimensions of each needed block.

These directives can be implemented by the following piece of code:

```java
// instantiate a DataContentSet of cardinality n
DataContent dataset = new DataContentSet(n){...}

// set a matrix rXc as view of the input data set
MatrixView mv = dataset.getMatrixView(r,c);

// instantiate a parallel iterator accessing the
// element of the view by blocks of dimension xXy
Iterator it = mv.getParIterator(new ByPointPattern(x,y));
```

The \texttt{mv} object provides an instance of \texttt{ParIterator} that implements the \texttt{Iterator} interface and that will provide the minimal units of access as a whole. In order to declare which elements of the view the iterator have to consider a singleton, an instance of \texttt{AccessPattern} is provided to its constructor: in this particular case a \texttt{ByPointPattern}(x,y) pattern is required, i.e. the access pattern implementing the access on a matrix view by blocks of dimension \(x \times y\).
9.2 Control primitives

Package \texttt{it.unipi.di.sonia.collectives} provides all the classes needed to instantiate and evaluate the primitives given by the framework. In particular, this package allows to instantiate a set of class implementing the primitives needed in structuring the application graph, plus a set of classes hidden to the user control but useful to the runtime system.

The classes implementing the primitives are the following:

- \texttt{class Fun} implementing the \texttt{Fun} primitive
- \texttt{class Apply} implementing the \texttt{Apply} primitive
- \texttt{class Seq} implementing the \texttt{Seq} primitive
- \texttt{class Pipe} implementing the \texttt{Pipe} primitive

The temporal operators \texttt{seqdo} and \texttt{pardo} have also been implemented. In fact, the former is given by the canonical sequential execution of instructions, naturally exploited by a processor running a program. The latter is implemented by combining threads and remote procedure calls (RMI invocations) on a cluster of workstations. In other words, they are implemented by framework through the evaluation process itself.

The set of classes needed by the user to implement its parallel program is completed by two classes representing the remote runtime media of the framework:

- \texttt{class Evaluator} implements the function $E$, thus evaluates an input application graph according to a coupled iterator. The service provided by this classes is intended to be distributed upon a cluster of workstations, i.e. at running time there are a number of \texttt{Evaluator} instance remotely available for evaluating an input (sub)graph.

- \texttt{class Context} implements the platform on which the whole application will run. Particularly, this class is responsible for looking up remote services and providing remote instances of \texttt{Evaluator} to the local machine.

9.2.1 Framework startup and class Evaluator

As mentioned above, the target architecture of the implementation framework is a cluster of workstations. At running time, an \texttt{Evaluator} object is running on each processor of the cluster waiting for a graph or a sub-graph to evaluate. The evaluation mechanism is provided through the \texttt{eval} method, thus an instance of \texttt{Evaluator} is binded on the RMI registry of all the remote hosts.

In order to make the binding possible, a class representing \texttt{Evaluator} objects must implement the interface \texttt{EvaluatorIntf} that is the interface through which any object in the network can refer to a remote instance of \texttt{Evaluator}.

The synopsis of the interface is the following:

\begin{verbatim}
public interface EvaluatorIntf extends Remote{
    public DataContent eval(DataContent adt, Collective graph)
        throws RemoteException;
    public DataContent eval(Iterator it, Collective graph)
        throws RemoteException;
}
\end{verbatim}

The interface itself extends \texttt{java.rmi.Remote} in order to inherit the capability to be binded on a RMI registry. The \texttt{eval()} method has two different signatures. Both of them mean that the compound primitive \texttt{graph} is evaluated taking as input the iterator \texttt{it} and the process produces the abstract data type resulted from the application of the input graph to the tasks provided by the iterator. However, the
difference between the two signature is that in the first one the `Iterator` instance has been created independently from the type of primitive represented by `graph` (i.e. it has been instantiated by invoking a factory method on a given view). In the second case, the iterator instance that will be coupled with the primitive `graph` is the one built by the `builder` of the primitive instance. In fact, in the framework runtime, the evaluation of a primitive is always preceded by the construction of the iterator type such primitive is expecting. In this way each primitive can be associated with its own type of iterator.

Thus, the second synopsis of method `eval` is justified in all those cases in which the user chooses to build the input iterator of the primitive by means of the related `builder`, instead of instantiating it by directly invoking the method `getParIterator()` or `getSeqIterator()` of the input view.

The linguistic mechanism through which this association is expressed is represented by the method

```java
public Iterator setIterator(DataContent d);
```

that all the primitives must implement. In a sense, this method “implements” the `builder` property and, although implementation of primitives will be explained in the next sections, we will give here the idea of its usage.

Let us take a look at the following code:

```
// defines input data as a matrix of dimension r x c
1. MatrixView av = (new DataContentSet(n)).getMatrixView(8,8);

// creates an iterator
2. Iterator outer_it = av.getParIterator(new ByPointPattern(4,4));

3. Context ctx = new Context("hostnameFile.txt");

4. Apply inner_ap = new Apply(new MyFunction(), ctx){
   
   public Iterator setIterator(DataContent adt){
      ArrayView inner_view = adt.getArrayView();
      Iterator inner_it = inner_view.getParIterator(new ArrayPattern(2));
      return inner_it;
   }

   }

5. Apply outer_ap = new Apply(inner_ap, ctx);

6. ctx.getEvaluatorInstance().eval(outer_it, outer_ap);
```

Lines 1-2 define a matrix view on the input data type and a related parallel iterator. Such iterator returns tasks of dimension 4 \( \times \) 4 because of the type of access pattern its constructor receives as input. Line 3 defines a context and line 4 instantiate the inner `Apply` but it is a specialized version of the primitive. In fact, its method `setIterator()` is overridden by a new implementation that “interprets” the input `DataContent` object (i.e. the block of rows coming from the evaluation of the outer `Apply`) as an array of 4 \( \times \) 4 elements. On such array a parallel iterator reading couples of elements is instantiated and provided as method result.

In the example above, each instance of the inner primitive can “restructure” the task of type `DataContent` it receives from the outer one: before starting the evaluation of `inner_ap`, the framework evaluation mechanism will invoke the method `inner_ap.setIterator(outer_it.current())` in order to prepare the right iterator for the primitive.

---

1The definition of `builder` is given in section 5.4
For completeness, we should mention that the same piece of code might also be written as follows:

1. `DataContent d = new DataContentSet(n);`

2. `Context ctx = new Context("hostnameFile.txt");`

3. `Apply inner_ap = new Apply(new MyFunction(), ctx){
   public Iterator setIterator(DataContent adt){
      return adt.getArrayView().getParIterator(new ArrayPattern(2));
   }
};`

4. `Apply outer_ap = new Apply(inner_ap, ctx){
   public Iterator setIterator(DataContent adt){
      return adt.getMatrixView(8,8).getParIterator(new ByPointPattern(4,4));
   }
};`

5. `ctx.getEvaluatorInstance().eval(d,outer_ap);`

In this case, also the outer primitive specializes (line 4) the instantiation of its iterator on the input abstract data type and the second signature of method `eval()` can be invoked on the `Evaluator` instance (line 5). Obviously, this code is more expensive than the first one because it requires to pass the whole input `DataContent` object as argument to the outer primitive constructor and to the `eval` method.

### 9.2.2 Class `Context`

In order to instantiate a graph of primitives and to proceed with its evaluation, a context, i.e. a collector of information regarding the executer framework, has to be set up. Such information is carried out by an object of type `Context` that a user has to create before instantiating any primitives and that is known to each primitive and to the `Evaluator` object.

The main role of the `Context` object is to look up all the `Evaluator` instances disseminated across the network and to enqueue them in a private structure. For this reasons, the constructor for `Context` objects requires as input the list of host names participating in the target architecture (the same host names on which instances of `Evaluator` have been binded). Such host addresses can be passed as an array of `String` or as the content of a text file whose name is passed as argument. Thus, they are the couple

```java
public Context(String filename)
public Context(String[] hostnamesList)
```

Each time a subgraph evaluation is required, methods

```java
public EvaluatorIntf getEvaluatorInstance();
public EvaluatorIntf getRemoteEvaluatorInstance();
```

can be invoked in order to get, respectively, a local or a remote instance of class `Evaluator` to which ask for a primitive evaluation.

Each time the `getRemoteEvaluatorInstance` method is invoked, the private collection of references is accessed and an instance of `EvaluatorIntf` is returned using a round-robin policy. Instead, each time the `getEvaluatorInstance` method is invoked, a new `Evaluator` instance is created on the local machine.

However, from the user's point of view, a local `Evaluator` object suffices to start the process. In fact, since the graph is the result of two or more nested primitives, the evaluation of the outer
control pattern will require the evaluation of the encapsulated one until the leaf of such tree of evaluations (a Fun primitive) is reached. Each time the evaluation of a primitive $C$ depends on the evaluation of its inner primitive (for example, Apply depends on the encapsulated primitive), the outer primitive asks its Context for a remote instance of Evaluator in order to distribute the computation, coherently with the pattern of control it represents.

Example 9.2.1 The Mandelbrot application consists in applying a function $f$ measuring a pixel color to all the complex value of an input stream. The involved control pattern is the Apply one while the stream of values can be represented in different manner, can be accessed punctually or by blocks etc. Since we are here interested in focusing on the control programming aspects, the representation of the input data won’t be treated.

The graph representing the parallel evaluation of the Mandelbrot set is given by an Apply primitive that will apply the given sequential function to each minimal access unit.

The code produced by the user is the following

```java
// define data access aspects
Iterator it = ...

// instantiate the context on the available hosts
Context ctx = new Context(hostList);

// call the Apply constructor
Apply ap =....

EvaluatorIntf evaluator = ctx.getEvaluatorInstance();

DataContent result = evaluator.eval(it,ap);
```

Looking at example 9.2.1, it is worth noting that we have expressed the control concerns of the application without specifying how data is represented, located, how it will be accessed and so on. We have completely described the control pattern (represented by an Apply primitive) by simply selecting a primitive and specifying to the Context object the list of host names participating in the distributed computation. Moreover, the result of the computation is an object implementing the DataContent interface. The actual implementation of the returned data depends on the type of the sequential functions really manipulating the data.

In the next sections, we will detail the set of primitives a user can instantiate to build complex control graphs and how such set can be easily extended. First of all, we will illustrate the interface each primitive must implement: the Collective interface.

### 9.2.3 The Collective interface

The Collective interface must be implemented, at least, together with java.io.Serializable (for remote transferring).

Collective exposes only three methods

```java
public interface Collective{
    public Collective[] innerPrimitives(); // returns the list of
    public DataContent eval(Iterator id); //evaluates the inner primitive
    public Iterator setIterator(DataContent adt); //sets its own iterator
}
```

The first method, `innerPrimitives()` returns an array containing the encapsulated primitives. In fact, as we will see in a while, a primitive can encapsulate one or more primitives (excepting a Fun object that returns itself).
The second method, \texttt{eval()}, is the method the framework calls for evaluating the primitive. Such method takes as argument an iterator encapsulating the portion of data the primitive has to handle. For instance, if the primitive is an \texttt{Apply} instance, the implementation of the method will distribute the tasks provided by the iterator depending on its type (sequential or parallel).

The third method implements the \textit{builder} property of the primitive. By default, all the primitives mentioned in this thesis instantiate through this method a sequential iterator allowing to access all the singletons of the given DataContent. The code implementing it is given by

\begin{verbatim}
public Iterator setIterator (DataContent d){
    return (new ArrayView(d)).getSeqIterator(new ByArrayPattern(1));
}
\end{verbatim}

A user who wants to assign a different behavior to the \textit{builder} of a primitive, simply has to override this method, as shown in section 9.2.1.

\subsection*{9.2.4 Class Fun}

Class \texttt{Fun} is an abstract class representing a sequential function to be executed. In order to instantiate a primitive of this type, the user has to extend the class \texttt{Fun} and to provide the implementation of the method

\begin{verbatim}
public DataContent eval(Iterator it);
\end{verbatim}

that is the method the framework will call for executing the sequential function. Since \texttt{Fun} can be executed locally or remotely, it implements the \texttt{java.io.Serializable} interface as well as \texttt{Collective}.

The instantiation of a \texttt{Fun} object is done in the expected way:

\begin{verbatim}
Fun f = new MyFunction();
\end{verbatim}

where, as an example, \texttt{MyFunction} looks like

\begin{verbatim}
public class MyFunction extends Fun{
    public DataContent eval(Iterator it){
        while(it.hasNext()){ // compute task
            // .......
        }
    }
}
\end{verbatim}

As we can see from the method \texttt{eval()} synopsis, the primitive takes an iterator as input variable. This type of argument doesn't appear so strange if one realize that the literature about programming techniques has widely recognized the correctness of using iterators to access linear data structures. In particular, it helps avoiding access violation of the collection on which it is built. In the contest of our programming model, this role is much more evident, since the iterator is not only an abstraction for enumerating elements of a collection. Our iterators also encapsulate a behavior, they mask the global input data set to all those primitives working on a portion of it, they are, in one word, the only media through which a primitive will access the input abstract data type.

In this sense, the iterator given as input argument to a \texttt{Fun} object encapsulates a DataContent set or a singleton DataContent (depending on its own type) as representative of the piece of data the \texttt{Fun} object will access.

\subsection*{9.2.5 Class Apply}

Class \texttt{Apply} represents the implementation of the homonym primitive.

The synopsis of this class provides a constructor that accepts an inner collective as argument.
public Apply(Collective inner_primitive, Context ctx)

Indeed, the usage of the `Apply` constructor and the instantiation of the related graph is simply summarized as follows:

```java
    // instantiate a context
    Context ctx = new Context(host_list);

    // instantiate the inner primitive
    Fun f = new MyFunction();

    // instantiate the outer primitive and global graph
    Apply graph = new Apply(f, ctx);
```

Differently with respect to the `Fun` primitive and all the others, such `Apply` depends also from the context being instantiated. Such context will become important in providing the run time primitive the `Evaluator` remote references.

### 9.2.6 Class `Pipe`

Class `Pipe` represents the implementation of the homonym primitive. The synopsis of this class provides a constructor that accepts an inner collective list as argument and a `Context` instance

```java
public Pipe(Collective[] primitives_list, Context ctx)
```

The following code portion illustrates the usage of such constructor within a main program. We suppose to have two `Fun` objects (implemented by the class `MyFunction1`) representing the first and the third stage of the pipeline and an `Apply` primitive nesting the sequential `Fun` object `MyFunction2`, representing the second stage of the pipeline.

```java
    // instantiate a context
    Context ctx = new Context(host_list);

    // instantiate the inner primitive
    Collective[] inners = new Collective[3];
    inners[0] = new MyFunction1();
    inners[1] = new Apply(new MyFunction2());
    inners[3] = new MyFunction1();

    // instantiate the outer primitive and global graph
    Pipe graph = new Pipe(inners, ctx);
```

This code portion also gives a first example of nesting. As it can be seen, each collective could be encapsulated by an outer collective by simply passing it as argument to the constructor.

### 9.2.7 Class `Seq`

Class `Seq` represents the implementation of the homonym primitive. The synopsis of `Seq` type constructor is similar to the previous one

```java
public Seq(Collective[] primitives_list, Context ctx)
```

A `Seq` primitive is built from a list of primitives (each representing one of the primitives to execute sequentially respect to the other) and a `Context` instance.

As it can be seen the construction of a `Seq` primitive is similar to the one given for instantiating a `Pipe`. 
CHAPTER 9. A PROTOTYPE IMPLEMENTATION

// instantiate a context
Context ctx = new Context(host_list);

// instantiate the inner primitive
Collective[] inners = new Collective[3];
inners[0] = new MyFunction1();
inners[1] = new Apply(new MyFunction2());
inners[3] = new MyFunction1();

// instantiate the outer primitive and global graph
Seq graph = new Seq(inners, ctx);

9.2.8 An extension: class Loop

Let us suppose we want to extend the set of primitives with a simple control primitive: Loop. From a functional point of view, a Loop primitive takes a graph as input and repeats its evaluation \( k \) times, where \( k \) is defined by the user. Thus, we can imagine a constructor of class Loop given by the following synopsis:

```java
public Loop(int k, Collective inner_primitive)
```

In order to allow the framework to recognize Loop as a new primitive, the implementation class must implement the interface Collective and the interface java.io.Serializable. As mentioned in section 9.2.1, primitives may be executed (i.e. their eval method may be invoked) by Evaluator instances distributed across the network, thus objects implementing primitives must have marshaling and unmarshaling capabilities that are inherited by the classes implementing interface java.io.Serializable from the Java framework.

We are now able to encode the first lines of our new primitive.

```java
public class Loop implements Collective, java.io.Serializable{
    private int numCycles; // number of iterations
    private Collective inner_p // the inner primitive to evaluate

    public Loop(int k, Collective inner_primitive){
        numCycles = k;
        inner_p = inner_primitive;
    }
}
```

Once the class constructor has been defined, we have to implement the methods required by interface Collective.

The method setIterator should return the iterator instance a Loop primitive would expect if evaluated by an outer primitive. In this case, since the eval method is a mere repetition of the inner primitive evaluation, the expected iterator for Loop coincides with the iterator expected by such primitive. Thus, the method is quite simple

```java
public Iterator setIterator(DataContent adt){
    return inner_p.setIterator(adt);
}
```

As mentioned before, the eval method is intended to be a cycle of invocations to the eval method of the encapsulated primitive, thus

```java
public DataContent eval(Iterator it){
    DataContent res = null;
```
The last method is quite obvious and returns an array of size 1 containing the only primitive encapsulated by the primitive.

```java
public Collective[] innerPrimitives(){
    Collective[] c = new Collective[1];
    c[0] = inner_p;
    return c;
}
```

In the following, the code related to class `Loop` is summarized.

```java
public class Loop implements Collective, java.io.Serializable{
    private int numCycles; // number of iterations
    private Collective inner_p // the inner primitive to evaluate

    public Loop(int k, Collective inner_primitive){
        numCycles = k;
        inner_p = inner_primitive;
    }

    public DataContent eval(Iterator it){
        DataContent res = null;
        for (int i = 0 ; i < numCycles; i++ ){
            res = inner_p.eval(it);
            it = inner_p.setIterator(res);
        }
        return res;
    }

    public Iterator setIterator(DataContent adt){
        return inner_p.setIterator(adt);
    }

    public Collective[] innerPrimitives(){
        Collective[] c = new Collective[1];
        c[0] = inner_p;
        return c;
    }
}
```

A main program using the primitive `Loop`, looks like the others we have already seen:

```java
import it.unipi.di.sonia.collectives.*;

public class UseOfLoop {
    public static void main (String argv []){
    ```
... // definition of data access

Context ctx = new Context(fileofhostname);

Collective graph = ......... // defines a composite, inner graph

int k=... // defines the number of iterations
Loop l = new Loop(k,graph); // instantiates a Loop primitive

EvaluatorIntf evaluator;
evaluator = ctx.getEvaluatorInstance(); // gets an Evaluator instance
evaluator.eval(l); // evaluates the graph

9.3 Development guidelines

The simple class written for extending the set of primitives as shown in the previous section, suggests that the system developer can focus exclusively on the control concerns. In the example above, all the concerns related to data are encapsulated into the Iterator item passed as argument to the new primitive eval method. Potentially, someone else could program the data concerns part, and who develops the primitives doesn’t need to know any detail.

The developer of a new primitive must simply be careful in applying some simple development rules, we can summarize as follows:

1. the new class must implement the interface Collective in order to be recognized as primitive by the framework
2. the new class must implement class java.io.Serializable in for allowing remote transferring
3. the constructor of the new primitive depends on its functionality. Generally speaking, a primitive can realize pure control flow or it can also handling data. A primitive that realize pure control flow is a primitive like Loop or Fun, and simply define how data flows through the graph without manipulating it. On the contrary, a primitive could handle data meaning that
4. if the new primitive gather or distribute tasks to other subgraph (i.e. it must demand pieces of work to remote Evaluator instances), an object of type Context must be passed as argument to the constructor. For example, Apply is a primitive that remotely evaluates its inner primitive for each input task. In order to contact the available remote Evaluator instances, it needs to know which Context has been created (because, as explained in section 9.2.2, such object is responsible for looking up and enqueuing the remote references of Evaluator objects).

On the other hand, a primitive like Loop, that realizes pure local control flow without relations with other processing elements, doesn’t need to know a specific context and, thus, its constructor doesn’t mention an object of type Context.

Definitively, in order to extend the set of primitives it suffices to know some basic rules about the framework runtime and its expected interfaces.

9.4 Examples

In this section we will detail the encoding of a subset of applications we have considered in chapter 8. In this section we will focus on the programming methodology and on the use of the abstraction
mechanisms provided by the framework. In the next chapter, we will detail some performance measures related to such examples and, in particular, we will have an actual feedback on the cost inequalities our computational cost system has defined in chapter 7.

Our programming model completely decouples data parallel concerns from control parallel concerns. In the following examples this fact will be more clear. In fact, we will demonstrate once the data parallel parts of our application has been fixed, the control part can be differently implemented, thus exposing different performance results and a different control structure. On the other hand, we will show how, once a control graph has been selected, the data parallel concerns can be differently managed, thus changing the data handling of the application but not its semantics.

### 9.4.1 Data parallel concerns

Our experiments start from the instantiation of a DataContentSet, initialized through $n$ randomly generated values. Since the random generation is not implemented by the class constructor, we have to extend the class by providing a new version of the method `init()` (as already explained in example 9.1.2).

On the input dataset, an array view is built. The view is associated with a pattern of access `ArrayPattern` of granularity `GrainBLOCKS`, given as input to the constructor of an iterator. This means that such iterator will read/write blocks of dimension `GrainBLOCKS` when it access the view. Since the iterator is the contact point between data parallel and control parallel concerns, its type depends on the particular control graph we are encoding.

Thus, the portion of code that handles data parallel concerns is the following:

```java
// instantiates a DataContentSet of cardinality n
DataContent dataset = new DataContentSet(n){
    public void init(){
        Random r = new java.util.Random();
        for (int i = 0 ; i < n ; i++ )
            this.setElementAt(new Integer(r.nextInt()),i);
    }
}

// instantiates an array view
ArrayView av = new ArrayView(dataset);

// instantiates a sequential iterator
Iterator its = av.getSeqIterator(new ByArrayPattern(GrainBLOCKS));
// ... or a parallel iterator....
Iterator its = av.getParIterator(new ByArrayPattern(GrainBLOCKS));
```

### 9.4.2 Apply(seqdo) implementation

The following code refers to the application graph $G$ analyzed in section 8.1.1. The graph implements the application of a sequential function composed by three sub-functions $f_1,f_2,f_3$ to all the elements of the input dataset. Thus, we are encoding an `Apply` primitive taking as input the composition of three `Fun` instantiation. The composition of these three sub-functions has been implemented by a `Seq` primitive encapsulating them. For simplicity we assume that each sequential function is an instantiation of class `MyFunction` we have detailed in section 9.2.4.

Thus, the application graph is encoded by the instructions below that follow those given for the implementation of the data parallel concerns.

```java
// instantiates a Context
ctx=new Context("hostfilename");
```
CHAPTER 9. A PROTOTYPE IMPLEMENTATION

// instantiates the three sequential primitives
Collective[] list = new Collective[3];
list[0] = new MyFunction();
list[1] = new MyFunction();
list[2] = new MyFunction();

// instantiates the composition of three primitives
Seq s = new Seq(list);

// instantiate the Apply of the composition
Apply apply = new Apply(s, ctx);

// gets the local Evaluator
EvaluatorIntf evaluator = ctx.getEvaluatorInstance();
DataContent result = evaluator.eval(its, apply);

This version of the application that applies a composition of MyFunctions to all the values of the input dataset, associates a sequential iterator to the Apply primitive. This means that, for each element provided by the iterator, Apply will ask the Context for a remote Evaluator instance that will apply an instance of Seq to the current portion of data.

9.4.3 seqdo(Apply) implementation

This application is the counterpart of the preceding one. As explained in section 8.1.1, we consider the composition of a three Apply primitives, each encapsulating a sequential function represented by a Fun object. The overall application applies a composition of three sequential functions to the input data set, as seen in the previous section, but the graph of control is completely different. In this version the outer primitive is represented by Seq taking as input a list of Apply instances. Each instance redefine its builder in order to create access its portion of data by blocks of grain GrainBLOCKS provided by a sequential iterator of array view.

Obviously, the parameters for the eval method need to be updated with the new outer primitive. Moreover, since each Apply defines its own iterator, we don’t need to define an iterator on the initial dataset as done in the previous example. As a consequence, the eval method will take as input a reference to the input dataset instead of an iterator instance.

Thus, the implementation of data parallel concerns given in section above, can be extended with the following code

// instantiates a Context
Context ctx=new Context("hostfilename");

//instantiates the three sequential primitives
Collective[] list = new Collective[3];

// instantiates the Apply primitives to compose
list[0] = new Apply(new MyFunction(), ctx){
    public Iterator setIterator(DataContent ds){
        return (new ArrayView(ds)).getSeqIterator(new ByArrayPattern(GrainBLOCKS));
    }
};

list[1] = new Apply(new MyFunction(), ctx){
    public Iterator setIterator(DataContent ds){
        return (new ArrayView(ds)).getSeqIterator(new ByArrayPattern(GrainBLOCKS));
    }
};

list[2] = new Apply(new MyFunction(), ctx){

9.4. EXAMPLES

```java
public Iterator setIterator(DataContent ds) {
    return (new ArrayView(ds)).getSeqIterator(new ByArrayPattern(GrainBLOCKS));
}
};

// instantiates the composition of the three Apply primitives
Seq s = new Seq(list);

// gets the local Evaluator
EvaluatorIntf evaluator = ctx.getEvaluatorInstance();
DataContent result = evaluator.eval(its, seq);
```

### 9.4.4 Apply(Pipe) implementation

The application proposed in this section is similar to the first one but instead of a Seq instantiation, the Apply primitive encapsulates a Pipe instantiation. The pipeline stages are represented by three Fun primitives each embedding a sequential function implemented by a MyFunction object. The code for building and evaluating this graph is very similar to the one detailed in section 9.4.2. Simply, we have substituted the instance of Seq with an instance of Pipe.

```java
// instantiates a Context
ctx = new Context("hostfilename");

// instantiates the three sequential primitives
Collective[] list = new Collective[3];
list[0] = new MyFunction();
list[1] = new MyFunction();
list[2] = new MyFunction();

// instantiates the composition of three primitives
Pipe p = new Pipe(list, ctx);

// instantiate the Apply of the composition
Apply apply = new Apply(p, ctx);

// gets the local Evaluator
EvaluatorIntf evaluator = ctx.getEvaluatorInstance();
DataContent result = evaluator.eval(itp, apply);
```

In order to realize a plain parallelism exploitation, we have coupled a parallel iterator with the Apply primitive. This means the all the elements provided by the iterator are accessible in a whole but they will be distributed in parallel to the remote instances of Evaluator executing the Pipe sub-graphs.

### 9.4.5 Pipe(Apply) implementation

This control graph represents the counterpart of the previous one and a little modification of the code provided in section 9.4.3. In fact, the outer primitive is represented by a Pipe primitive instead of a Seq. The Evaluator object will start the evaluation of Pipe coupled with the sequential iterator its, on the local machine: for each task farmed out by its, a private thread inside the class Pipe will be activated: all threads will ask for the execution of the stages composing the whole pipeline (array list) to a remote instance of Evaluator. Each chain will be applied on one of the tasks provided by the outer iterator it, provided as input argument to the related thread.

```java
// instantiates a Context
Context ctx = new Context("hostfilename");
```
// instantiates the three sequential primitives
Collective[] list = new Collective[3];

// instantiates the Apply primitives to compose
list[0] = new Apply(new MyFunction(),ctx){
   public Iterator setIterator(DataContent ds){
      return (new ArrayView(ds)).getSeqIterator(new ByArrayPattern(GrainBLOCKS));
   }
};

list[1] = new Apply(new MyFunction(),ctx){
   public Iterator setIterator(DataContent ds){
      return (new ArrayView(ds)).getSeqIterator(new ByArrayPattern(GrainBLOCKS));
   }
};

list[2] = new Apply(new MyFunction(),ctx){
   public Iterator setIterator(DataContent ds){
      return (new ArrayView(ds)).getSeqIterator(new ByArrayPattern(GrainBLOCKS));
   }
};

// instantiates the composition of the three Apply primitives
Pipe p = new Pipe(list,ctx);

// gets the local Evaluator
EvaluatorIntf evaluator = ctx.getEvaluatorInstance();
DataContent result = evaluator.eval(its,p);

9.4.6 A concrete class

We propose here the encoding of the class ApplyOfSeq implementing subsection 9.4.2 in order to
given an idea on how a complete implementation looks like in our framework.

import it.unipi.di.sonia.adt.ByArrayPattern;
import it.unipi.di.sonia.adt.DataContent;
import it.unipi.di.sonia.adt.DataContentSet;
import it.unipi.di.sonia.adt.Iterator;
import it.unipi.di.sonia.collectives.Evaluator;
import it.unipi.di.sonia.collectives.Context;
import it.unipi.di.sonia.collectives.Apply;
import it.unipi.di.sonia.collectives.Seq;
import it.unipi.di.sonia.collectives.Collective;

public class ApplyOfSeq{
   public static void main(String argsp[]){

      DataContent dataset = new DataContentSet(n){
         public void init(){
            Random r = new java.util.Random();
            for (int i = 0 ; i < n ; i++ )
               this.setElementAt(new Integer(r.nextInt()),i);
         }
      }
   }
ArrayView av = new ArrayView(dataset);

Iterator its = av.getSeqIterator(new ByArrayPattern(GrainBLOCKS));

Context ctx=new Context("hostfilename");

Collective[] list = new Collective[3];
   list[0] = new MyFunction();
   list[1] = new MyFunction();
   list[2] = new MyFunction();

Seq s = new Seq(list);

Apply apply = new Apply(s,ctx);

EvaluatorIntf evaluator = ctx.getEvaluatorInstance();
DataContent result = evaluator.eval(its,apply);
}
}

The implementation of the class representing the sequential function MyFunction has been given in section 9.2.4.

9.5 Remarks

The abstraction offered by our framework has allow also to encode some more complex application as, for example, the one solving the $k$-parity problem on a parametric number of variables by using programming techniques coming from the field of genetic programming. At the moment we are still working on it, because its inherently unbalanced workload requires some accuracy in the scheduling of the executors.
Chapter 10

Experimental results

In this chapter we will present some experimental results about the performance of the implementation framework and about the predictive power of our rewriting rules. At the end of the chapter, the reader could object that the examples we have provided are not too much significant because they represent simple cases of application graph. However, we would like to remind that the implementation framework is not intended to be an exhaustive, efficient and definitive programming platform that concretely realizes all the features offered by the semantic model. The implementation framework has been developed step by step as soon as the semantic framework was defined, aiming at obtaining an immediate feedback on two fundamental aspects: the usability of the approach and encouraging efficiency results. For usability we mean, particularly, how much easy is the encoding of a parallel program through the provided abstraction mechanisms. We have detailed this aspect in sections 9.1, 9.2.8. Moreover, in section 9.3 we have also provided a development guideline on how the framework could be easily extended.

In the following sections we will verify the completion time of some of the applications theoretically studied in chapter 8. A reference implementation of such application has been given in chapter 9. Now, we aim at concretely verify if the transformations suggested by our semantic framework will lead to more efficient application graphs.

For each sample application we will provide the completion time measured on a cluster of workstation (technical details are shown in Tab.10.1), efficiency and scalability results. Moreover, some of them will be compared in order to verify the inequalities related to computational costs studied in chapter 8.

<table>
<thead>
<tr>
<th></th>
<th>Ram Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Gb</td>
<td>Gb Memory</td>
</tr>
<tr>
<td>2</td>
<td>Hard-Disks, 18 Gb each</td>
</tr>
<tr>
<td>1</td>
<td>Intel(R) Pentium(R) III Mobile CPU (800 MHz)</td>
</tr>
<tr>
<td>32 KB</td>
<td>Cache L1</td>
</tr>
<tr>
<td>3</td>
<td>Ethernet Pro 100</td>
</tr>
<tr>
<td>24</td>
<td>remote nodes</td>
</tr>
</tbody>
</table>

Table 10.1: Characteristics of cluster pianosa.di.unipi.it

10.1 The experiments

The experiments we are going to discuss are relative to the applications detailed in section 9.4.2 to section 9.4.4. Data handling is the same in all the implementations: to each element of an input DataContentSet of size n, three sequential functions are applied. On the dataset an array view is built. Such view provides a sequential iterator of blocks of dimension GrainBLOCKS each. For each implementation we will provide completion time, scalability and efficiency measures.
All the experiments refer a case study in which the dataset size is composed by \( n = 4800 \) integer values and the grain of each block accessed by the outer iterator is \( \text{GrainBlocks} = 600 \) values. The completion time of the sequential algorithm executed on a machine of the cluster has been estimated to be
\[
T_{\text{seq}} = 407653 \text{ ms}
\]

The sequential algorithm iterates for each element of the input data block a loop of 300000 iterations, each applying a non trivial multiplication between trigonometric functions and randomly generated values of type \text{double}.

and it will be compared with the completion of the parallel versions that will be labeled \( T(n) \), where \( n \) represents the number of available hosts or processing elements (PEs) of the underlying target architecture.

We have performed a whole set of experiments. The results shown here have been selected to give a precise idea of the results achieved without being concerned with different details of the non parallel part of the application.

### 10.1.1 \text{Apply(\text{seqdo}) performance}

The first experiment regards the application detailed in section 9.4.2. It refers to an \text{Apply} primitive who distribute the task provided by the outer sequential iterator to a set of remote \text{Seq} instances, each encapsulating the composition of three \text{Fun} primitive.

Fig.10.1 shows the completion time over a range of 2 to 24 machines.

![Figure 10.1: Completion time for the \text{ApplyOfSeq} application graph](image)

As it can be seen, the completion time is quite near the ideal one from 2 to 4 host. For 4 to 6 host, there is a sort of constant completion time that decreases from 6 to 8 and that becomes constant after 8 processors available. Such behavior can be better appreciated in Fig.10.2 and Fig.10.3 showing the scalability (that stop growing after 8 processors) and efficiency (between 1.01 and 0.70 from 1 to 10 processors, but in case with 6 processors) of the application, respectively.

Scalability stops growing after 8 processors because the new available hosts are not exploited since we have only 8 partitions instances of sub-graph to gather. The efficiency remains over 70% from 1 to 10 processors, but in case of 6 processors, thus showing a good usage of the resources.

\footnote{This numbers have been chosen as a trade-off to show the features of the prototype without going to evaluate “extreme” cases, i.e. those dominated by either communication or computation times.}
Such behavior, in this particular instantiation of the problem, the actual processor needed by the application for a parallel execution is 8, i.e. the number of blocks the outer iterator provides and, as a consequence, the number of Seq replication that the Apply primitive activates. From 2 to 6 the number of processors is lesser than the actual need, thus on some processors the Evaluator object performs more than one evaluation of the Seq subgraph. But, with 2 and 4 processors (divisor of 8), the workload is still balanced between the processors, while with 5 and 6 processors, one or two of them evaluates more than the other, thus causing the delay. This behavior is confirmed by Fig.10.4 in which on the x-axis we have a number of processors that coincides for each value of the coordinate with the outer iterator size. In this case the completion time is better than the ideal one because the parallelism is fully exploited thanks to the plain availability of hosts.

This example clearly demonstrates how choosing the smart partitioning of the input dataset can heavily influence the overall performance of the application. In the same time, as we have seen
in the previous chapters, the abstraction power of iterators is so strong that a minimal effort is required to modify the iterator size and, consequently, the overall graph configuration to a more efficient one.

10.1.2 **seqdo(Apply)** performance

This experiment is related to the application detailed in section 9.4.3. It refers to a composition of Apply primitives (embedded by Seq) who distribute the task provided by the sequential iterators built by their own builder to a set of remote Fun instances, each representing a sequential function object.

Fig.10.5 shows the completion time over a range of 2 to 24 machines.

As seen in the previous example, for a number of blocks higher than the number of remote
hosts available, the completion time is worst with respect to the ideal one. But also in this case we can demonstrate that the problem is related to the unbalanced distribution of the workload.

![Completion time graph](image1.png)

Figure 10.6: Completion time for SeqOfApply when \( size(it) = \#PE \)

In fact, Fig. 10.6 shows what happens by executing the same application of a number of host equal to the number of blocks that are generated by the outer iterators of each Apply instantiation. Also in this case, the completion is quite near the ideal one because the parallelism is fully exploited thanks to the plain availability of hosts.

10.1.3 Apply(Pipe) performance

The ApplyOfPipe application sketched in section 9.4.4 has produced the performance results depicted in Fig.10.7. The application regards an outer Apply primitive that replicates the execution of a pipeline of three sequential objects (Fun instantiations), taking as input a parallel iterator.

![Completion time graph](image2.png)

Figure 10.7: Completion time for the ApplyOfPipe application graph
CHAPTER 10. EXPERIMENTAL RESULTS

The chart shows a quite regular behavior of the application with a growing completion time after 20 processors threshold. Looking at the scalability chart in Fig. 10.8 we can observe that performance degrades slowly starting from 8 processors (that is, in fact, the number of sub-graph the Apply instantiates).

The efficiency is shown in Fig. 10.9 and it keeps over 80% from 2 to 16 processors.

10.1.4 Pipe(Apply) performance

In Fig.10.10 the performance results related to the application graph illustrated in section 9.4.5 are shown. It refers to a pipeline of Apply primitive, each replicating the embedded Fun instantiation for each element provided by a sequential iterator.

As for its counterpart in the previous section, this application exploits a completion time quite
regular with respect to the ideal case and shows a scalability degradation around 8 hosts (see Fig. 10.11. Again, the efficiency keeps quite high until 20 processors as evidenced by the chart in
10.2 Rewriting rules exploitation

Objective of this section is to demonstrate that our experiments confirm the prediction about completion time provided by the semantic framework. In particular we have formally verified that:

1. \texttt{ApplyOfSeq} is cheapest (or faster) than \texttt{SeqOfApply} (rule 8.1.1)
2. \texttt{ApplyOfPipe} is cheapest (or faster) than \texttt{PipeOfApply} (rule 8.2)
3. \texttt{ApplyOfSeq} is cheapest (or faster) than \texttt{ApplyOfPipe} (rule 8.3)
4. \texttt{SeqOfApply} is cheapest (or faster) than \texttt{PipeOfApply} (rule 8.4)

The comparison offered by case 1 is depicted in Fig. 10.13 and the curves confirm the prediction given by the costs system.

The comparison offered by case 2 is depicted in Fig. 10.14 and also in this case, the curves confirm the prediction provided by the costs system.

Case 3 is shown in Fig. 10.15. Here, the inequality between \texttt{ApplyOfSeq} and \texttt{ApplyOfPipe} given by rule 8.3 appears to be not confirmed when more that 9 hosts are used. However, it must be pointed out that the performance of \texttt{ApplyOfSeq} is strictly related to the parameters of this particular instance of the program. As mentioned before, it can use always eight hosts or less, because eight is the size of the input iterator: when the number of hosts is lesser than eight, an unbalanced workload augments the completion time (in particular, for a number of PEs between 5 and 6); when the number of hosts is greater than 9, the new available hosts are not exploited because the parallelism won’t increase if the iterator size doesn’t augment.

As a consequence, augmenting the number of available hosts doesn’t improve the performance of the program. On the contrary, as the number of hosts increases, the performance of \texttt{ApplyOfPipe} improves.

On the other hand, the rules given in chapter 7, don’t take into account the actual number of available hosts: in fact, the cost prediction assumes that all the needed hosts are available. As a consequence, when cases of unbalanced workload arises, the prediction could fault.

The observation above holds also in case 4, depicted in Fig. 10.16. Again, the current instantiation of \texttt{SeqOfApply} suffers of an unbalanced workload caused both by a number of hosts lesser than the iterator size and, after 9 hosts, by the inability of exploiting all the hosts (parallelism degree) available.
10.3 Comparisons between estimated and actual computational costs

In this section we will give an idea of the predictability power of our costs system by comparing the curves of actual completion time with the curves traced by the theoretic model. We will just two cases belonging to the set of rules in chapter 7, that are expressive do show the strength and weakness of the approach.

In section 8.1.1 we have estimated the theoretical computational cost of a graph applying in parallel a composition of sequential functions to an input dataset. The formal representation of the graph is given by

\[ G = \text{Apply}(\text{Fun}(f_1; \ldots; f_m)) \]
CHAPTER 10. EXPERIMENTAL RESULTS

ApplyOfSeq vs ApplyOfPipe completion time

SeqOfApply vs PipeOfApply completion time

while its estimate cost is equal to

$$\$ (G) = c_v \times (n + 1) + c_b \times n + \sum_{i=1}^{n} c_f$$ (10.1)

where $c_v$ represents the cost of a local method invocation, $n$ represents the iterator size, $c_b$ represents the cost of building an iterator on a remote machine for accessing a certain portion of data and $c_f$ is the computational cost of the composition of function.

In chapter 9 we have given the class ApplyOfSeq an implementation of the application and we are now able to compare the completion time of ApplyOfSeq with the one predicted by our cost system, once we have assigned a numeric value to the symbolic costs in 10.1. Since the invocations of methods and functions on a local machines don’t influence the overall performance of the distributed application, we can simplify 10.1 by assuming $c_v \approx 0$. Since $n$
represent the iterator size, in the actual instantiation of the problem \( n = 8 \). The value \( c_b \) is related to the cost paid for transferring a block of 600 integers (one portion of the input dataset) to a remote Evaluator instance that will build an iterator on top of the block and, then, applying a sub-graph on it. Our experiments have estimate \( c_b = 39 \text{ms} \). Moreover, since the sequential composition involves three instantiation of the same function implemented by class \( \text{MyFunction} \), we have estimated that \( c_f = 3 \times c_{\text{MyFunction}} \). Here, \( c_{\text{MyFunction}} \) is the cost of evaluating the sub-graph \( \text{MyFunction} \) coupled with an iterator that provides 600 integers. We have estimated that \( c_{\text{MyFunction}} = 169989 \text{ms} \).

By substituting the symbolic values with the actual numbers, the completion time predicted by the semantic framework for the \texttt{ApplyOfSeq} application is equal to

\[
T_{th} = 39 \times 8 + \sum_{t=1}^{8} (3 \times 169989) = 408048 \text{ms}
\]

Chart 10.17 provides a comparison between the actual completion time, the predicted completion time and the completion time in the case of full parallelism exploitation. The chart demonstrate two facts: first of all, the theoretical completion seems to be quite “optimistic” with respect to the actual performance of the application. As explained before, this happens because the implementation suffers of unbalanced workloads when the number of PEs is lesser then 8 (i.e. the iterator size); on the other hand, at least 8 machines are used to exploit parallelism. Since the theoretical formula doesn’t take into account such kind of delays, there is a natural distance between the two curves.

![ApplyOfSeq vs theory](chart10.17.png)

Figure 10.17: Actual vs theoretical completion time of \texttt{ApplyOfSeq}

On the other hand, when \texttt{ApplyOfSeq} exploits the maximum possible parallelism (i.e. when it uses all the available hosts re-dimensioning the iterator size), theoretical completion and actual completion are very close.

The same behavior can be observed by comparing \texttt{SeOfApply} completion time and its theoretical estimation. The latter is given by assuming \( c_v \approx 0 \), \( n = 8 \), \( m = 3 \), \( c_b = 39 \) and \( c_f = 169989 \text{ms} \) in

\[
\$\{\texttt{Seq}([\texttt{Apply(Fun}(f)]; \ldots; \texttt{Apply}(f))]) = c_v \times m \times n + m \times c_b \times n + n \times \sum_{t=1}^{n} c_f,
\]

Thus

\[
T_{th} = 3 \times 39 \times 8 + 8 \times \sum_{t=1}^{3} 169989 = 408672
\]
The chart in Fig. 10.18 demonstrates that the theoretical estimation is optimistic with respect the real one but, on the other hand, its very close if the application can exploits maximum parallelism also by redefining the input iterator.

![ApplyOfSeq vs theory](image)

**Figure 10.18:** Actual vs theoretical completion time of SeqOfApply

In fig. 10.19, the chart demonstrates the case of the PipeOfApply application, already exposing a good completion time and scalability during the performance test.

![PipeOfApply vs theory](image)

**Figure 10.19:** Actual vs theoretical completion time of PipeOfApply

In this case, actual curve and theoretical curve are much closer than the other cases. The slight difference between them can be justified by overheads introduced by the framework implementation that can be certainly improved. As an example, the scheduler dispatching remote Evaluator instances, at the moment, implements a round robin policy that contributes to the workload unbalance. However, we should recall the experimental nature of the framework that has not been provided with an accurate optimized implementation.
10.4 A case study: Genetic Programming

Evolutionary algorithms are heuristic strategies that mimics the process of natural evolution described by Darwin [Dar] in order to solve global search problems. This kind of algorithms assumes that the individuals that better adapt to the external environment, are the ones who have greater chance to survive and generate offsprings. More pragmatically, an evolutionary algorithm is represented by a population of individuals that evolve from one generation to the other, by means of a set of mechanisms similar to sexual reproduction and gene mutations. In this way the heuristic behavior focuses on those research areas in which better solutions can be found.

A canonical classification given for evolutionary algorithms distinguishes between genetic algorithms [Van04] 2 and genetic programming [Koza, Kozb].

A genetic algorithm (GA) [Hol] is represented by

- a finite population of individuals (each encoded by a string of bits) of size \( M \);
- an adaptation function, called fitness, providing an adaptation capability measure of an individual. In other words, the fitness function estimates how “good” a solution is and provides an indication about the best candidates for reproduction;
- a set of genetic operators (crossover and mutation) that evolve the current population in the next one;
- a termination criterion establishing when the algorithms stops.

GAs are capable of solving many problems and simple enough to allow solid theoretical studies. Nevertheless, the fixed-length string representation of individuals that characterizes them is difficult, unnatural and constraining of a wide set of applications. In these cases the most natural representation for a solution is a hierarchical computer program rather than a fixed-length character string. But above all, GA representation schemes do not have any dynamic variability: the initial selection of string length limits in advance the number of internal states of the system and limits what the system can learn.

This lack of representation power is overcome by genetic programming (GP) [Koza, Kozb], which operates with very general hierarchical computer programs. In other words, GP considers individuals as tree structures. These structures are perfectly capable of capturing all the fundamental properties and features of modern programming languages (conditionals, iteration, etc.).

In synthesis the GP paradigms breeds computer programs to solve problems by executing the following steps:

1. Generate an initial population of computer programs. The set of all the possible structures that GP can generate is the set of all the possible trees that can be built recursively from a set of function symbols \( F = \{ f_1, \ldots, f_n \} \) (used to label internal tree nodes) and a set of terminal symbols \( T = \{ t_1, \ldots, t_m \} \) (used to label leafs).

**Example 10.4.1** Given the set of functions \( F = \{ +, - \} \) and the set of terminals \( T = \{ x, 1 \} \), a legal GP individual is represented by the tree in Fig.10.20

![Figure 10.20: A legal GP individual](image)

---

2 *Evolutionary strategies and evolutionary programming* are other two distinctions not much investigated by the scientific community, actually
2. Iteratively perform the following steps until the termination criterion (for instance, a maximum number of generations) has been satisfied:

- execute each program in the population and assign it a fitness value according to how well it solves the problem
- create a new population by applying the following operations:
  - Probabilistically select a set of computer programs to be reproduced, on the basis of their fitness (selection). There are several methods by which selection can be performed.
  - Copy some of the selected individuals, without modifying them, into the new population (reproduction)
  - Create new computer programs by genetically recombining randomly chosen parts of two selected individuals (crossover). Standard GP crossover begins by independently selecting one random point in each parent (it will be called the crossover point for that parent). The crossover fragment for a particular parent is the subtree rooted at the node lying underneath the crossover point. The first offspring is produced by deleting the crossover fragment of $T_1$ from $T_1$ and inserting the crossover fragment of $T_2$ at the crossover fragment of $T_1$. The second offspring is produced in a symmetric manner.
  - create new computer programs by substituting randomly chosen parts (the so called mutation point) of some selected individuals with new randomly generated ones (mutation).

3. the best computer program appeared in any generation is designated as the result of the GP process at that generation. This result may be a solution (or an approximate solution) to the problem.

In [Koza], Koza defines a set of problems that can be considered as belonging to the class of “typical GP problems” and which have the relevant feature of being simple to define and to apply to GP. One of such problems is the so called Even $k$-parity problem. In the next section we will give an overview of its implementation principle. In section 10.4.2 we will introduce parallel and distributed solutions for GP and in section 10.5 we will show the results obtained by some parallel implementations of the even $k$-parity problem.

### 10.4.1 A benchmarks for GP: the Even $k$-parity problem

The goal of this problem is to find a boolean function of $k$ boolean arguments that returns $true$ if an even number of its boolean arguments evaluates to true, and that returns $false$ otherwise. Each individual in the space of solutions is represented by a tree whose internal nodes are labeled by boolean operators $F = \{\text{and, or, not}...\}$ and the leaves are labeled by one of the $k$ terminal symbols in $T = \{x_1, \ldots, x_k\}$. Let us consider the case $k = 2$. A boolean function $f(x_1, x_2)$ perfectly solving this problem must respect the optimal truth table 10.2 in which every line represents a fitness case. For each individual, i.e. for each boolean expression represented by a tree in the space of solutions, the fitness value is given by the number of fitness cases in its truth table that matches the corresponding fitness cases in the optimal truth table.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$f(x_1, x_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 10.2: Truth table of the optimal individual for the even 2-parity problem
For instance, let $e = \text{or}(\text{and}(x_1, x_2), \text{and}(x_1, x_1))$ an individual of the current population. Its truth table is represented in Tab.10.3. Since two cases matches with respect to the optimal table (in detail, the first and the third case), the fitness value of $e$ is equal to $2^3$.

Thus, solving a solution for the even $k$-parity problem means finding an individual (i.e. a boolean function) whose raw (standardized) fitness is the maximum (the minimum) possible within a certain number of generations.

### 10.4.2 Parallel and distributed Genetic Programming

Genetic programming and, generally speaking, evolutionary algorithms need of very large population in order to exploit good convergence. Moreover, the fitness evaluation for a single individual implies the evaluation of a certain training set. These two aspects lead to a great computational effort. In addition, when considering genetic programs, each tree could be very complex; in this case the computational effort regards both temporal and spatial dimension and the latter could exceed the memory space of a single machine. Thus, a parallel implementation of this kind of algorithms is need, also considering distributed environments that could allow to overcome memory space problems. On the other hand, parallelism is implicit in nature, where individuals compete simultaneously for survival and the evaluation of the fitness (the more expensive phase of the algorithm) is independent for each individual and can be done in parallel.

There are several levels at which an evolutionary algorithm can be parallelized, the most important being the level of the fitness evaluation and the level of the population.

The first level leads to a **global model** in which, as in a sequential program, a single population is always kept but the fitness evaluation is executed in parallel for each individual. The computational load is distributed among the available processors while selection, crossover and other genetic operators are applied sequentially by considering the whole population.

In the **island model**, the population is divided into subpopulations, called islands, one per processor distributed. The implementation of such model requires the creation of random subpopulations onto each single processor, while genetic operators and fitness evaluation are performed at local level. A variation of such model requires the migration of a certain number of elements from one subpopulation to another, aiming at introducing diversity in the islands and performing a better convergence.

In the **grid model**, each individual $t$ is substituted in the next generation by a new element generated by applying crossover or mutation operators to neighbors of $t$. The selection reduced to a local neighborhood allow a very efficient parallel implementation (due to a lower communication overhead).

### 10.5 Even $k$-parity problem implementation

We have implemented two different versions of the even $k$-parity problem both inspired to the **island model**. The two versions have same behavior but different control graph structures. Since we are not yet supporting a distributed data implementation as native data type, we will need to create the set of individuals in a centralized way and to distribute it onto the processors in order

---

3This measure is called **raw fitness**. Generally, it is associated with the **standardized fitness** that is equal to the maximum raw fitness, i.e. the maximum number of possible matches (in this case 4), minus the actual raw fitness.
to emulate single subpopulations. Once a subpopulation has been assigned to a processor, all the
genetic operators are locally applied.

We have enriched the model by inserting a migration phase, in which a percentage of individuals
in each subpopulation migrates to a neighbor island. Conversely, each subpopulation accepts the
same percentage of immigrants coming from a neighbor island.

Before entering in a detailed analysis of the two versions, we will show their similarities.

**Population initialization** Each individual is represented as a tree object of type `LabeledTree`.
The internal nodes of each tree are labeled by objects of type `FunctionalSymbols` (representing
the set of boolean operators $F = \{ \text{and}, \text{or}, \text{not} \}$), while the leaves are labeled by objects of type
`Terminals` representing the $k$ variables of the problem, i.e. the set of terminals $T$.
The set of individuals (population) is represented by an instance of `DataContentSet` and is gen-
erated by means of the so called *grow initialization* method: a random symbol is selected with
uniform probability from $F$ to be the root of the tree. Then, $n$ nodes (where $n$ is the arity of the
selected functional symbol) are selected with uniform probability from the set $F \cup T$ to be its sons.
Finally, for each function symbol among these $n$ nodes, the method is recursively applied, i.e. its
sons are selected from the set $F \cup T$, unless this symbol has a depth equal to $d - 1$. In the latter
case, its sons are selected from $T$.

**Distribution of subpopulations** As mentioned before, we are not yet supporting a distributed
implementation of the interface `DataContent` but or abstractions on data are sufficiently powerful
to provide an emulation of distributed data. Once a centralized population has been created by instancing a `DataContentSet` of `LabeledTree` objects, it can be given to the main control graph as input.
The outer primitive of the control graph is an `Apply` instance whose `builder` defines on the input
abstract data type (i.e. the whole population) an `ArrayView` of $h$ elements, where $h$ is the number
of desired subpopulations (i.e. available processors). On top of such view a parallel iterator is
created and, thus, used by the primitive in order to access the $h$ subpopulations as a whole and
to distribute them. At this point onto each processor a subpopulation is allocated and the genetic
part of the program can be evaluated.

What distinguishes the two versions of the even $k$-parity problem we will show in the next
sections is the implementation of the such genetic part encapsulated by the outer `Apply` primitive
described above.

**Migration** The last common point between the two versions of the problem is represented by
the implementation of the migration phase.
The idea is that, after a certain number of generations, in each subpopulation (i.e. a partition of
the whole population) a percentage of individuals has to leave the subpopulation while a the same
percentage of immigrants has to join it.
By exploiting the mechanisms provided by the view abstraction, the migration can be represented
as a *shift* of elements applied on an array view of the whole population.
The effect of applying a *shift* operator of $w$ steps on an array view of length $n$ is that all the
elements are shifted by $w$ positions, modulo $n$. Thus, $\text{shift} : V \times \text{int} \to V$ is the type of such
operator.
As an example, if $v = < t_1, \ldots, t_n >$ is the configuration of the array before applying $\text{shift}(v, 2)$,
then $v = < t_{n-1}, t_n, t_1, t_2, \ldots, t_{n-2} >$ will be the resulting configuration.
How can the *shift* operator implement the migration phase of the even $k$-parity problem? The
idea is that once a given population $P = \{ t_1, \ldots, t_n \}$ partitioned into subpopulations $P_1, \ldots, P_h$ has
evolved through a number of generations, an array view is applied on $P$. Then, the *shift* operator is
applied on such view providing a new view of the population in which all the individuals are shifted
by $w$ positions. At this point the evolution restarts by instantiating an `Apply` primitive
that distributes the individuals provided by such new view, as explained above.
As a consequence, when the Apply primitive applies an array view of length $h$ onto the shifted population in order to emulate the $h$ islands, some individuals falling into the $i$-th island before the shift, will fall into the $i+1$-th island, for each $i \in [1, h]$.

The implementations of the even $k$-parity problem are closed by an additional evaluation of the fitness values related to the last population obtained, in order to sequentially select the best individual.

In the following section we will detail the implementations given to the even $k$-parity problem and we will provide some experimental results in terms of absolute and compared performance.

### 10.5.1 Impl2 version

The first implementation given to the problem is represented in class Impl2 by the following graph:

\[
\text{graph}_2 = \text{Apply}(\text{Loop}(\text{Apply}(\text{Fun}(kfitness;select;cross;mute), interval)))
\]

where $kfitness$ is a sequential function evaluating the fitness values of a set of individuals (i.e. LabeledTree objects); $select$ is a sequential function applying to its input subpopulation the selection of those individuals able to reproduce, to mute or to cross with other individuals: each individual is marked with a flag indicating the evolution to be applied. $cross$ is the sequential function applying the crossover to each pair of individuals marked as “crossable” into its input subpopulation; $mute$ represents the sequential function applying a mutation to each tree marked as “mutable” into its input subpopulation.

The outer primitive of this graph is the Apply primitive distributing the input population onto the available processors. Then, a Loop instance is applied to each subpopulation. The loop iterates $interval$ times its body, where $interval$ represents the number of generations to produce before applying a migration phase. The body of the loop consists in an Apply primitive that applies to a subpopulation a sequential function represented by the composition of $kfitness, select, cross$ and $mute$.

![Even 12-parity completion time popSize=15000 depht=10](image)

**Figure 10.21: Actual vs theoretical completion time of Impl2**

Fig.10.21 shows the completion time of the application adopting \text{graph}_2 as core control structure. The chart shows the comparison with a sequential implementation indicating the ideal completion time with respect to the number of available processors.

The experiments refer instances of the problem with 12 variables, 15000 individuals and trees whose depth is, at most, 10.
10.5.2 Impl3 version

The second implementation given to the problem is represented in class Impl3 by the following graph:

\[
\text{graph}_3 = \text{Apply}(\text{Loop}(
    \text{Apply}(\text{Fun}(k\text{fitness}));
    \text{Apply}(\text{Fun}(\text{select};\text{cross};\text{mute}))),
    \text{interval}),
)\]

Again, the \text{Apply} primitive distributes the subpopulation. After that, to each subpopulation a \text{Loop} primitive iterating \text{interval} times is applied. The body of such loop is represented by the composition of two \text{Apply} instantiations: the first one applies the \text{kfitness} sequential function to each individual of the subpopulation; the latter applies the composition of \text{select}, \text{cross} and \text{mute} to each individual of the subpopulation. Fig.10.22 shows the completion time of the application adopting \text{graph}_3 as core control structure. The chart shows the comparison with an sequential implementation indicating the ideal completion with respect to the number of available processors. Again, the experiments refer instances of the problem with 12 variables, 15000 individuals and tree depth of 10.

As it can be seen, it doesn’t result to be a very efficient solution, and, in a sense, it can’t surprise. In fact, the fitness evaluation is the more expensive phase but it is not the case of the function \text{(select;cross;mute)}. So, the overhead introduced in \text{graph}_3 by separating such function from \text{kfitness} and by adopting a new instance of \text{Apply} to perform it, heavily influences the overall performance.

A clear evidence of this influence will be provided by comparing implementation Impl3 with Impl2.

10.5.3 Computational costs comparison

In this paragraph we will demonstrate that

1. \text{graph}_2 comes from \text{graph}_3 by simply applying the transformation rule 8.1.1 illustrated in chapter 8 that demonstrates the functional equivalence between \text{graph}_2 and \text{graph}_3
2. the prediction about the completion time comparison about Impl2 and Impl3 is successfully confirmed by collected experimental results.

As it can be seen by comparing the two implementations, Impl2 results from Impl3 by applying rule 8.1.1, whose demonstration assesses that \( \text{Apply}(\text{Fun}(f_1;\ldots;f_n)) \) is functionally equivalent to \( \text{Apply}(\text{Fun}(f_1));\ldots;\text{Apply}(\text{Fun}(f_n)) \). In our case,

\[
\text{graph}_3 = \text{Apply}(\text{Loop}((\text{Apply}(\text{Fun}(k\text{fitness}));\text{Apply}(\text{Fun}(\text{select};\text{cross};\text{mute}))),\text{interval}))
\equiv \{ \text{by applying rule 8.1.1} \}
\text{Apply}(\text{Loop}(\text{Apply}(\text{Fun}(k\text{fitness};\text{select};\text{cross};\text{mute}))),\text{interval}))
= \text{graph}_2
\]

Moreover, from rule 8.1.1 we also know that it should be

\[ S(\text{Impl2}) \leq S(\text{Impl3}) \]

Fig.10.23 confirms such prediction. The chart compares the completion time of Impl2 and Impl3 clearly showing the better behavior of Impl2.

![Figure 10.23: Impl2 vs Impl3 completion time](image)

Anyway, it should be pointed out that the ones shown are not the best implementation a smart programmer would write. In fact, \( \text{graph}_2 \) can be further improved by swapping the \text{Loop} primitive with the inner \text{Apply} and, then, by collapsing the two outer \text{Apply} instances. We have called Impl1 such implementation of the problem.

Computationally, this means that rather than instantiating for each subpopulation an \text{Apply} for each iteration (i.e. generation), the population is initially distributed by a unique instance of \text{Apply} while the iterations and the genetic operators are locally applied on each processor.

The transformation swapping an \text{Apply} with a \text{Loop} and the one collapsing two nested \text{Apply}s (under certain conditions) are not treated in this thesis but its effect can be appreciated in fig.10.24 where Impl1 completion time is compared with the one related to Impl2.

At the end, we can conclude the following chain of transformation:

\[ \text{Impl3} \rightarrow \text{Impl2} \rightarrow \text{Impl1} \]

where the relation \( \rightarrow \) indicates a transformation whose left-hand member "reduces" to a more efficient one.
10.6 Conclusions

The results exposed in this chapter allow us to point out some considerations. First of all, separating data parallel from control parallel concerns by means of our set of abstractions seems to be a successful approach. In fact, managing data is more comfortable, also when it has to be partitioned and distributed across a network. A flat set of data can be structure many and many times into the same application without requiring particular programming efforts. Violations caused by erroneous data accesses are avoided by the use of iterators that encapsulate accessing rules as well as (parallel or sequential) behavior.

The design of an application graph is a matter of selecting and composing a set of primitives exposing a very simply interface. However, the set of primitive as well as the set of data abstraction can be extended both in the programming model domain (i.e. by inserting new views and/or new primitives) and in the semantic framework (i.e. by providing for new abstraction mechanisms semantic definitions and/or inference rules).

The high level of abstractions in describing data and control concerns, doesn’t affect the performance of the applications, as demonstrated by the charts shown above. The most scalability problems are not related to the programming model design, nor to the abstraction mechanisms assumed. Rather, they can be related with the actual implementation of the framework or with parts of it (as, for example, the scheduler of remote executors). A better design of the framework that was initially born as an experimental platform for collecting feedbacks about the theoretical work, will probably expose a better performance behavior.

By comparing the expected completion time with the actual completion time for a subset of applications, we have demonstrated that we can rely on the computational cost predictions provided by the costs system. The accuracy of the formulae can be certainly improved and we have seen that a first scenario to be investigated is the one considering unbalanced workloads during the parallel execution.
Chapter 11

Conclusions and future work

In this thesis we have proposed a new programming model in which parallel applications are built by keeping orthogonal data and control parallel concerns thought a set of abstraction mechanisms. A clear separation of such concerns evidenced by the fact that they can be described in a separated and independent way is the first contribution of this thesis.

Moreover, we have given a formalization of the model, pointing out that the evaluation of a parallel application is described by a sequence of transformations on the application graph involving abstract mechanisms for both expressing data access and control concerns. A further contribution given by the thesis is a clear, unambiguous specification of the semantics associated to each abstraction mechanisms as well as the introduction of a computational costs system provided by the semantics itself. Such cost system allow us statically associate a cost to the evaluation (i.e. the execution) of a graph of primitives by referring the cost of each component primitive.

At the moment, we are able

- to structure parallel application by simply selecting control primitives each embedding a precise control behavior

- to encapsulate data access details through iterator abstractions that are given as input to the evaluation process of the graph of primitives. From the user point of view this means an easy encoding of a parallel application also in case of control-intensive applications in which data have to be often exchanged. From the framework point of view encapsulation of data accesses means the possibility to optimize the underlying implementation and prevent the user from violation errors.

- to establish if the user defined graph of primitives is functionally equivalent to a cheapest one and, as a consequence,

- to suggest a potentially source-to-source graph rewriting.

We have presented some experimental results demonstration the expressive power of our abstraction and, on the other hand, the reliability of the computational cost system.

A first sketch of the whole thesis has been published in [CD04]; the formal representation of view/iterator/primitives abstractions has been accepted at [Cam05b]; the computational costs system has been introduced in [Cam05a] and it is going to be presented at the third international workshop on High-Level Parallel Programming and Applications (HLPP 2005), in July.

New efforts for future works can be invested in different directions, as suggested by the results offered by this thesis. The set of abstractions for describing both data and control parallel concerns can be further extended. New views and access patterns can be introduced in order to differentiate data handling. As an example, it could be very interesting to involve the development of some “irregular” view structure as for example tree view and/or graph view. On the same level, introducing new patterns of access can contribute to enlarge the spectrum of application that can
be encoded into the framework. As an example, patterns describing minimal unit of access related by some kind of functional dependencies with neighbor units could be an interesting developing challenge.

On the control concerns side, we have to extend the set of primitives by introducing new, more interesting and more powerful types. As an example, we have to introduce primitives describing irregular patterns of control (as for example, divide and conquer, broadcast, all-to-all communication and so on) that will allow to write more complex and more “difficult” (for the parallelism point of view) applications.

The abstraction mechanism represented by the iterator type can be further studied in order to find a way of defining the “optimal” iterator for a given graph. Such formal and practical definition could provide an alternative way of optimizing the application performance by handling data access rather than the control structure.

Probably, the more promising part of the model on which our efforts will concentrate is represented by the computational cost system. Our approach, based on assigning a cost to a graph starting from the cost of its component primitives, has demonstrated to be feasible and practically realizable. However, since there are many variables that can influence the execution of a parallel application, the computational cost system can be refined in order to be more precise. However, through our system of inference rules we have proposed a new way through which reasoning on the evaluation of a graph of control that consists in inspecting its characteristics by means of a set of semantics descriptions.

On the other hand, the experiments we have done on a class of applications that often in the literature have been presented as case study for graph rewritings, have demonstrated the feasibility of our approach.

A great challenge for the future, in addition to all the improvements we have cited above, is represented by the extension of the model to new and more complex platforms as Grid [FK99, La02, M.B02] architectures, in which problems related to dynamicity, discovery of resources and security will arise.
Bibliography


