Algorithmic Concept Recognition support for Skeleton Based Parallel Programming *

Beniamino Di Martino
Facoltà di Ingegneria - Dipartimento di Ingegneria dell’ Informazione
Second University of Naples - Italy
beniamino.dimartino@unina.it

Anna Bonifacio
Covansys Italia S.p.a. - Rome - Italy

Abstract

Parallel Skeletons have been proposed as a possible programming model for parallel architectures. One of the problems with this approach is the choice of the skeleton which is best suited to the characteristics of the algorithm/program to be developed/parallelized, and of the target architecture, in terms of performance of the parallel implementation. Another problem arising with parallelization of legacy codes is the attempt to minimize the effort needed for program comprehension, and thus to achieve the minimum restructuring of the sequential code when producing the parallel version. In this paper we propose automated Program Comprehension at the algorithmic level as a driving feature in the task of selection of the proper Parallel Skeleton, best suited to the characteristics of the algorithm/program and of the target architecture. Algorithmic concept recognition can automate or support the generation of parallel code through instantiation of the selected Parallel Skeleton(s) with template based transformations of recognized code segments.

1 Introduction

Decades of activity in parallel programming development has shown that the “parallel structure” of most programs can be classified into a limited number of categories, that can be viewed as representative of the basic ways to organize a parallel computation [1, 2, 14]. This notion of parallel structure of a program has been referred with the term Parallel Programming Paradigm, or Parallel Skeleton.

Structured composition of Parallel Skeletons has been proposed as a possible programming model for parallel architectures, and structured coordination notations and languages, such as PCN [7] and Fortran M [8] and, at a higher level, $P^3L$ [3] and SCL [4], have been designed based on this approach. The structured composition of Skeletons’ approach can help in achieving efficiency, ease of use, reusability and portability in parallel programming. Each Skeleton could be univocally characterized in terms of its performance behaviour on each target architecture of interest; for instance, a performance model for Paradigm constructs of the $P^3L$ language is specified in [11].

One of the issues arising with this approach is the mapping of the algorithm, or of an already developed sequential program to be parallelized, to a Skeleton which is best suited to the characteristics of the algorithm/program, and of the target architecture, in terms of performance of the parallel implementation. This issue arises especially when trying to use different Skeletons for handling different phases of the algorithm/program, or when trying to compose Skeletons in a hierarchical way. But, even when the algorithm belongs to a well defined algorithmic class, the choice of the mapping Skeleton could depend on minor aspects of the given concrete algorithm, which are nevertheless relevant with respect to performance issues, and thus crucial for the Skeleton’s selection. An additional complication is present when parallelizing an already existing sequential program, especially for the so-called “legacy code”; in this case a compelling target, as important as the performance requirements, is the minimum restructuring of the sequential code, in order to minimize the effort required in understanding the code, and in developing the parallel version.

We believe that automated Program Comprehension at the algorithmic level can be a driving feature in the task of selection of the proper Parallel Skeleton, best suited to the
The Algorithmic Recognition strategy

The recognition strategy we have designed is based on hierarchical parsing of algorithmic concepts. The recognition process is represented as a hierarchical abstraction process, starting from an intermediate representation of the code at the structural level in which base concepts are recognized; these become components of structured concepts in a recursive way. Such a hierarchical abstraction process can be modeled as an hierarchical parsing, driven by concept recognition rules, which acts on a description of concept instances recognized within the code.

The concept recognition rules are the production rules of the parsing: they describe the set of characteristics that allow for the identification of an algorithmic concept instance within the code.

The characteristics identifying an algorithmic concept can be informally defined as the way some abstract entities (subconcepts), which represents a set of statements and variables linked by a functionality, are related and organized within a specific abstract control structure. By “abstract control structure” we mean structural relationships, such as control flow, data flow, control and data dependence, and calling relationships.

More specifically, each recognition rule specifies the related concept in a recursive way, by means of:

- a compositional hierarchy, recursively specified through the set of subconcepts directly composing the concept, and their compositional hierarchies;
- a set of conditions and constraints, to be fulfilled by the composing subconcepts, and relationships among them (which could involve subconcepts at different levels in the compositional hierarchy, thus not only the direct subconcepts).

Attributed grammars [10] have been selected as formalism for the specification of the recognition rules of the hierarchical concept parsing. It allows for a good expressiveness for the specification of the hierarchy, the constraints and the relationships among subconcepts (not only direct subconcept, but at any level in the hierarchy) and the specification of the general hierarchical concept parsing mechanism.

The properties and relationships which characterize the composing concepts have been chosen in such a way to privilege the structural characteristics respect to the syntactic ones. We have decided to give the data and control dependence relationships a peculiar role: they become the characteristics that specify the abstract control structure among concepts. For this purpose, they undergo an abstraction process during recognition, such as the concept abstraction process. This abstraction has been represented by
the introduction of the notions of abstract control and data dependence among concept instances; such relationships are specified in a recursive way. The set of abstract data and control dependence relationships is produced within the context of the concept parsing process, and is explicitly represented within the program representation at the algorithmic level.

The direction of the concept parsing has been chosen to be top-down (descendent parsing). This choice is motivated by the particular task of the recognition facilities in the framework of the parallelization process. Since we are interested in finding instances of parallelizable algorithmic patterns in the code, an algorithmic recognition of the whole code is not mandatory: thus a top-down parsing (demand-driven), which leads to partial code recognition, is suitable, and allows for a much deeper pruning of the search space associated with the hierarchical parsing than the bottom-up approach.

The base concepts, starting points of the hierarchical abstraction process, are chosen among the elements of the intermediate code representation at the structural level. The code representation at the structural level (Abstract Base Program Representation) is thus a key feature that affects the effectiveness and generality of the recognition procedure; we have chosen the Program Dependence Graph [6] representation, slightly augmented with syntactical information (e.g. tree-like structures representing expressions for each statement node) and control and data dependence information (edges augmented e.g. with control branch and data dependence level, type, dependence variable). Two main features make this representation suitable for our approach: (1) the structural information (data and control dependence), on which the recognition process relies, is explicitly represented; (2) it’s an inherently delocalized code representation, and this plays an important role in solving the problem of concept delocalization.

An overall Abstract Program Representation is generated during the recognition process. It has the structure of a Hierarchical PDG (HPDG), reflecting the hierarchical strategy of the recognition process. As long as the parsing process proceeds and more and more abstract concepts are recognized, they are represented as nodes in increasingly higher layers of the HDPG. The nodes of this graph are connected by two kind of edges. The hierarchy edges connect each node representing a concept to the lower layer nodes representing its subconcepts. The graph structure determined by this kind of edges represents the hierarchy of abstraction; this structure is generally a tree, except in the case of shared concepts. The dependence edges link together nodes that have abstract control and data dependence relationships between them. Note that, during the recognition process, dependence edges for the newly created abstract concept nodes are inherited from those of the composing subconcept nodes in a way that is characteristic of each concept.

The described approach to algorithmic pattern recognition permits successful handling of the main problems arising for automated program comprehension, within the context of imperative languages like Fortran and scientific computation. These are:

- **Syntactic variations**: they consist in different possible implementations of the same algorithmic pattern, which result in the same control and data flow structure.
- **Implementation Variations**: an algorithmic concept represent an abstract algorithmic functionality. An instance of it can thus be an implementation of different concrete algorithms, which determine the same functionality.
- **Data structure implementations**: instances of a same algorithm can present different structure, depending on the particular implementation of the abstract data structure on which the algorithm acts.
- **Optimizations**: an algorithm can present several optimizations, related to the characteristics of the executing architectures, which determine more or less deep differences among the several instances of the algorithm.
- **Delocalization**: an algorithmic instance could not be implemented within a contiguous portion of code, but could be spread through the program text, depending on the freedom allowed by its dependence structure.
- **Instances overlapping**: instances of one or more distinct algorithmic patterns could share code portions (this is often a particular form of optimization).

The syntactic variation problem is solved by: (1) characterizing inter-statement level concepts with the structural properties of control and data dependence; (2) representing program expressions by means of abstract structures and performing symbolic analysis of them by means of functions which manipulate the abstract structures. The delocalization problem is solved by the characteristics of the Abstract Program Representation which: (1) is based on an inherently delocalized structural representation (Program Dependence Graph); (2) has a global scope of visibility, so that the active rule can attempt to match all instances of concepts already recognized, at every level of the abstraction. Although this characteristic in principle increases the complexity of the process, the systematic use of control and data dependence relationships to characterize concepts allows the application of rules to be driven by the locality typically present in the source program. In this way complex-
ity can be maintained at an acceptable level, without constraining the delocalized recognition capability. The implementation variation problem is solved by the backtracking feature of the recognition process. More specifically, backtracking allows the specification of one concept by means of multiple rules; each rule specifies a different algorithmic implementation of the same concept. However, backtracking has also its drawbacks. If on one hand it makes the recognition procedure more powerful and general, on the other hand it makes the search complexity to grow exponentially with the code size. Nevertheless, as we have observed above, both the top down approach and the summarization of derived subconcepts within HPDG nodes should prune the search space considerably making practical the analysis of non trivial pieces of code. Finally, the overlapping implementation problem is solved by the global scope of visibility of the representation, and by the fact that the parsing mechanism does not restrict the use of a subconcept to one rule, thus allowing the recognition even in presence of shared concept instances.

An important consequence of the features just discussed is the independence from restructuring techniques, that modify the original code before and during the recognition process to deal with delocalized code and implementation variations. This means that our approach does not need a canonical form for concept implementations (even though pre-applied restructuring transformations could still be useful in certain situations to speed up the recognition process).

3 ALCOR: ALgorithmic COncepts Recognizer

In this section the design and prototypical implementation of ALCOR (ALgorithmic COncepts Recognizer) is described; it implements the methodology for the algorithmic recognition outlined in the previous section. The input code is Fortran or C, and the recognition of algorithmic concept instances within it is performed without any supervision from the user. The code recognition is nevertheless partial (only the concept instances specified in the Alcor’s inferential engine are recognized) and limited to the functional (algorithmic) level: concepts related to the application domain level of the source code aren’t taken into consideration, in order to design a completely automatic procedure. These limitations are nevertheless irrelevant, due to the purpose of recognition: to drive parallelization strategies, the algorithmic level comprehension is sufficient, and it can be applied on partial portions of the code. Results of the recognition, that is the set of recognized parallelizable algorithmic pattern instances and their characterizing attributes, are presented to the user, by means of a suitable graphical user interface. In figure 1 a scheme representing the architecture of the tool, its components, and the main interactions among them, is presented.

The main Alcor’s modules are the Structural Analysis, the Semantic Analysis and the GUI Components. The GUI component coordinates the interaction among the user and the other components. The user selects the input program and drives the activation of the Structural and Semantic Analysis modules, through the AST, PDG and Algorithmic Concepts panels. The Structural Analysis module, builds up the basic program representation described in the previous section, composed by a syntax tree and associated program dependence graph, which are stored in the AST and PDG databases depicted in fig. 1. Graphical representations of AST and PDG are provided by the AST and PDG panels. The user can graphically browse the syntax tree and program dependence graphs and, by clicking on their nodes, it is possible to highlight the corresponding code portions. The PDG information, together with suitable syntactical information (e.g. tree-like structures representing expressions for each statement node) are then passed to the Semantic Analysis module, by converting them into a set of prolog facts, representing the Abstract Base Program Representation. The recognition, that is the hierarchical parsing process, is performed by the Inferential Engine, which applies the production rules of the parsing (implemented by Prolog clauses and stored in the Concept Recognition Rules database) to the set of terminal, non terminals and rela-
tionships among their attributes represented in the Abstract Program Representation. Result of the concept parsing is the production of Prolog facts representing the recognized concept instances (whose parameters represent the values of the associated attributes), and Prolog facts representing the abstract dependence relationships among them. These facts are stored within the Concept Instances Database, and they represent the Abstract Representation of the code inspected. Last task of the GUI Component is the representation of the concept instances recognized, and the attribute values. One key feature to be represented is the hierarchical composition of the recognized concepts. This is represented, through the GUI’s Algorithmic Concepts’ Panel, in a form of a graph, where each node represents a subconcept instance: by clicking on one of these nodes it is possible to highlight the code portions implementing the corresponding subconcept instance, and the nodes of the graph representing its own subconcepts. The construction of this graph is performed jointly by the Hierarchy Scanner, a syntactical analyzer which decomposes the representation of the concept hierarchy (in the form of a Prolog compound term) in tokens, and by the Hierarchy Parser, a syntax-directed translator which translates the Prolog representation of the hierarchy and builds the graph. The structural analysis module has been built by using the COCKTAIL compiler construction toolkit [9]. The syntax tree and PDG representations have instead been built through the Ast toolkit and Puma [12] tools. The implementation of the data flow and data dependence analysis phases has been achieved utilizing the Omega library [13], a package for linear systems solution and for set computation and manipulation. First order logic programming, and SWI Prolog in particular, has been utilized to implement the semantic analysis module and to perform the hierarchical concept parsing, thus taking advantage of SWI Prolog’s deductive inference rule engine. Finally, the GUI module has been implemented using the Java environment and the Java’s Swing graphical library.

4 How algorithmic concept recognition can support skeleton based parallel programming

In this section we exemplify, with the help of a case study, how Program Comprehension can support the selection of Parallel Skeletons and their instantiation. In particular we describe how the recognition of a particular algorithmic pattern (Divide and Conquer) in codes implementing sorting algorithms (Quick Sort) or optimization algorithms (Branch & Bound), can drive the selection of specific Parallel Skeletons (Tree Computation and Processor Farm) and their instantiation with recognized code fragments of the sequential codes. The Parallelizable Algorithmic Pattern we consider represents the class of Divide and Conquer algorithms: examples are n-body problems, Branch and Bound algorithms and sorting algorithms. All of them share a divide and conquer resolution strategy, corresponding to the traversal of a (abstract) tree; the nodes of this tree are the subproblems obtained by dividing the parent problem, and the tree traversal represents the recursive operation of dividing and conquering the problem. Of course, the kind of problem, and so the actual nodes of the tree, depends on the particular algorithm. The (iterative) Divide and Conquer algorithmic pattern is described in an informal way in fig. 2:

The features that characterize this algorithmic pattern concern the way some abstract functions (split problem, push a subproblem onto stack, pop problem from stack, set current problem to other subproblem, etc.) are related and organized into a specific control structure.

We present now two programs which, despite being an implementation of completely different algorithms, both represent an instance of the D&C algorithmic pattern.

The program excerpt in Fig. 3 is an implementation of a quick sort algorithm, where a list is ordered by splitting and recursively ordering the sublists (the divide and conquer strategy of this algorithm, corresponding to the traversal of an abstract tree, is implemented in an iterative way). The second program, represented in Fig. 4, implements binary branch-and-bound search, and includes inputting of initial data and outputting of the final result.

Even though the two programs perform completely different functions, they both share an iterative divide-and-conquer resolution strategy corresponding to the traversal of a binary tree. Each node of the tree corresponds to a sublist to be sorted in the first program, and to a search subspace in the second program. Both programs share the basic control and data structures that are outlined in the two figures by means of annotations on the right of the code. More specifically, the tree traversal is performed by execut-

Figure 2. Parallelizable Algorithmic Pattern of (Iterative) Divide and Conquer.
Program Quicksort

```fortran
module Quicksort
contains
  type stacknode
    integer left
    integer right
    integer node%left
    integer node%right
    type (stacknode), pointer :: node%next
  end type stacknode

  integer M
  integer N

  logical notempty

  variable R

  integer i
  integer j
  integer k

  integer head

  type (integer), pointer :: node
  type (stacknode), pointer :: top

  subroutine quicksort
    integer left
    integer right
    integer i
    integer j
    integer K
    integer flag
    integer RR
    real RR
  end subroutine quicksort
end module Quicksort
```

Figure 3. Program (Fortran 90) implementing the "Quick Sort" algorithm.

Figure 4. Program (C) implementing the "Branch and Bound" algorithm for a discrete optimization.
ing a loop that, at each iteration, processes a data structure representing the current problem and derives two subproblems from it. One subproblem become the current problem, while the other is pushed onto a data structure implementing a stack. Alternatively, if the current problem satisfies a given condition, some processing is possibly done and a new current problem is popped out from the stack. The loop terminates when the stack is empty. Note that the way the stack and the problems are represented in the two programs are quite different.

The key part of the Divide and Conquer pattern, the tree traversal, could be performed in parallel according to one of two Parallel Skeletons: tree computation and processor farm.

The first PS consists of a set of processes connected by communication channels according to a binary tree structure. Each process receives a problem to be solved, tests for a condition and then either splits the problem into two subproblems that are sent to two child processes, or does some processing and returns a result to its parent. When a process sends two subproblems, it remains waiting for a reply from each child, then combines these replies and sends back the new result. In practice a number of questions (the depth of the tree of processes, if processes have to be created dynamically or the entire tree must be instantiated statically, etc.) have to be answered before a working program can be generated. However, they do not depend on the particular nature of the computation to be parallelized, but rather they are part of the PS and can be solved once and for all in the context of the paradigm itself. Figure 5 illustrates the PS through a graph specifying how processes communicate (Fig. 5a), and a skeleton code describing the behaviour of the component processes (fig. 5b).

The second PS, processor farm, consists of a coordinator process and a set of worker processes that act as slaves of the coordinator. In a processor farm, the coordinator decomposes the work to be done into subproblems and assigns a different subproblem to each worker. Upon receipt of a subproblem, each worker solves it and returns a result to the coordinator. Again some details have to be defined before the PP can become a working program, and slightly different organizations can be selected for the processor farm (for instance workers may or may not be allowed to communicate with each other). However, even in this case, these issues do pertain to the PS definition and can be entirely dealt with in the paradigm context. The communication graph and the skeleton codes of the coordinator and worker processes composing the processor farm PS are illustrated in Fig. 6a,b,c.

Which of these two paradigms to choose depends on the underlying architecture, but also on features of the particular divide and conquer algorithm, and so on the recognition of relevant features in the code implementing it. Indeed, if the problem decomposition phase is the costliest part of the algorithm, and so is better performed in parallel, the tree computation is more convenient. In any case, care has to be taken that subproblems are generated with nearly the same amount of associated work, leading to a well balanced tree computation that fits the process structure of the paradigm. Conversely, if the split phase is quite cheap, whereas the subproblems generated might correspond to very different amounts of work, the processor farm paradigm appears to be best suited, since on the one hand splitting subproblems sequentially does not represent a significant penalty, and, on the other hand, it is quite easy to embed a load balance policy in the coordinator that can keep workers busy most of the time.

These considerations lead us to chose the tree computation PS for the quick sort program, and the processor farm for the branch-and-bound one. This because the split phase in the quicksort is the costliest part of the algorithm. Moreover, unless a worst-case list is given in input, each split phase generates two subproblems with nearly the same associated work. Conversely, in the branch-and-bound case the split phase is quite cheap, whereas the subproblems generated might correspond to very different amount of work, owing to the presence of a pruning strategy to reduce the search space. Finally, in branch-and-bound, pruning is partly performed on the basis of a global value (the current optimum) that is kept updated and made available to all workers. In the processor farm paradigm, this can be guaranteed if the coordinator collects local optima from workers, and returns them the updated values of the global optimum.

The instantiation of the skeletons of the PS processes through fleshing of the skeletons of the chosen paradigm with code excerpts extracted from the original code is shown in fig. 5c (for the tree computation and quick sort) and fig. 6d,e (for the processor farm and branch-and-bound). In both cases, almost no change to the skeleton and to the sequential code to be inserted is required to generate the final parallel version of the quicksort algorithm. The only modifications concern operations such as recv "problem" from parent in the instantiation of the tree computation to the quick sort algorithm, which must be expanded in several simpler communications to cope with the structured nature of the problem description. Similarly, in the instantiation of the processor farm skeleton to the branch-and-bound algorithm, in the worker skeleton code, the when clause specifies that sequential code representing the body of the cycle must be modified so that a particular action takes place every time the "threshold" is set to "test pruning value".
Figure 5. The tree computation Parallel Skeleton: (a) communication graph; (b) skeleton code of a tree node process; (c) instantiation of the skeleton with code fragments of the quick sort program.
Figure 6. The processor farm Parallel Skeleton: (a) communication graph; (b) skeleton code of the coordinator process; (c) skeleton code of the worker process; (d,e) instantiation of the skeletons with code fragments of the branch and bound program.
5 Concluding remarks

In conclusion, we are sketching a procedure which, based on the automated algorithmic recognition performed by a program comprehension tool such as Alcor, can support, or automatically perform, code parallelization according to the skeleton-based approach. When a parallelizable algorithmic pattern is found by Alcor, it could be possible to exploit suitable information associated to the corresponding skeleton(s) to control subsequent steps of the recognition process. For instance, if we are analysing the branch-and-bound program of figure 4, after the iterative divide-and-conquer pattern has been identified, we know that two different PSs can be chosen to perform parallelization. We then try to find out if the split phase is cheap or expensive, or if some pruning condition is contained in the code (or both), in order to decide which paradigm is best suited for the concrete program given in input. It is worth noting that the Skeleton selection behaves like a sort of control input with respect to the recognition procedure. That reduces the complexity of the recognition process during the search towards identification of those algorithms that represent meaningful patterns with respect to parallelization. The parallel code generation can be performed by a code generator that receives in input a description of the selected Skeleton and the hierarchical description of the input program. The code generator builds the output parallel program starting from the instantiation of the parallel skeleton. Instantiation of PSs consists of extracting, from the original input, those code segments corresponding to parts of the skeleton that have been left generic. As we have observed in the previous example, in some cases this would require a certain degree of code transformation to integrate sequential code in a parallel, distributed memory framework. Once more, it is apparent the importance of exploiting skeleton’s information during the recognition process. Indeed, if some skeleton requires nontrivial transformations for sequential code integration, such as that reported in fig. 6d,e, the corresponding information can be easily added to the skeleton’s description stored in the PP database. This information may help the recognizer to identify just those particular concepts that are needed to correctly perform the mentioned transformations.

References


