View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

USING CLIPS IN THE DOMAIN OF KNOWLEDGE-BASED MASSIVELY PARALLEL PROGRAMMING

Jiri J. Dvorak
Section of Research and Development
Swiss Scientific Computing Center CSCS
Via Cantonale, CH-6928 Manno, Switzerland
Email: dvorak@cscs.ch

Abstract

The *Program Development Environment* PDE is a tool for massively parallel programming of distributed-memory architectures. Adopting a knowledge-based approach, the PDE eliminates the complexity introduced by parallel hardware with distributed memory and offers complete transparency in respect of parallelism exploitation. The knowledge-based part of the PDE is realized in CLIPS. Its principal task is to find an efficient parallel realization of the application specified by the user in a comfortable, abstract, domain-oriented formalism. A large collection of fine-grain parallel algorithmic skeletons, represented as COOL objects in a tree hierarchy, contains the algorithmic knowledge. A hybrid knowledge base with rule modules and procedural parts, encoding expertise about application domain, parallel programming, software engineering, and parallel hardware, enables a high degree of automation in the software development process.
In this paper, important aspects of the implementation of the PDE using CLIPS and COOL are shown, including the embedding of CLIPS with C++-based parts of the PDE. The appropriateness of the chosen approach and of the CLIPS language for knowledge-based software engineering are discussed.

1 INTRODUCTION

Massive parallelism is expected to provide the next substantial increase in computing power needed for current and future scientific applications. Whereas there exists a variety of hardware platforms offering massive parallelism, a comfortable programming environment making programming of distributed-memory architectures as easy as programming single address space systems is still missing. The programmer of parallel hardware is typically confronted with aspects not found on conventional architectures, such as data decomposition, data and load distribution, data communication, process coordination, varying data access delays, and processor topology. Approaches to a simplification of parallel programming that have been used previously, in particular for shared-memory architectures, are transformational and compile-time parallelization [Pol88, ZC90]. However, such code-level parallelization is extremely difficult for distributed architectures. Additionally, only a limited, fine-grain part of the opportunities for parallel execution within the program can be detected, and the results are dependent on the programming style of the programmer. The detection of conceptual, coarse-grain parallelism found in

many applications from natural sciences and engineering is in general too complicated for current parallelization techniques.

Our *Program Development Environment* PDE [Dvo93, DDR94] is a tool for programming massively parallel computer systems with distributed memory. Its primary goal is to handle all complexity introduced by the parallel hardware in a user-transparent way. To break the current limitations in code parallelization, we approach parallel program development with knowledge-based techniques and with user support starting at an earlier phase in program development, at the specification and design phase.

In this paper we focus on the expert system part of the PDE realized in the CLIPS [GR94] language. At first, a short overview of the PDE is given. The representation and use of knowledge is the topic of section 3. This is followed by sections showing aspects of external interfaces, problem representation, and embedding with C++. The important role of an object-oriented approach will be elaborated in these parts. A discussion of results, the appropriateness of the CLIPS language, and the current state of the development concludes the paper.

2 OVERVIEW OF THE PDE

The basic methodology of programming with the PDE consists of three steps [DR92]:

1. Problem specification using a domain-specific, high-level formalism
2. Interactive refinement and completion of the specification (if needed)
3. User-transparent generation of compilable program code

According to this three-step approach to the programming process, the program development environment consists of the three functional components shown in Fig. 1. In a typical

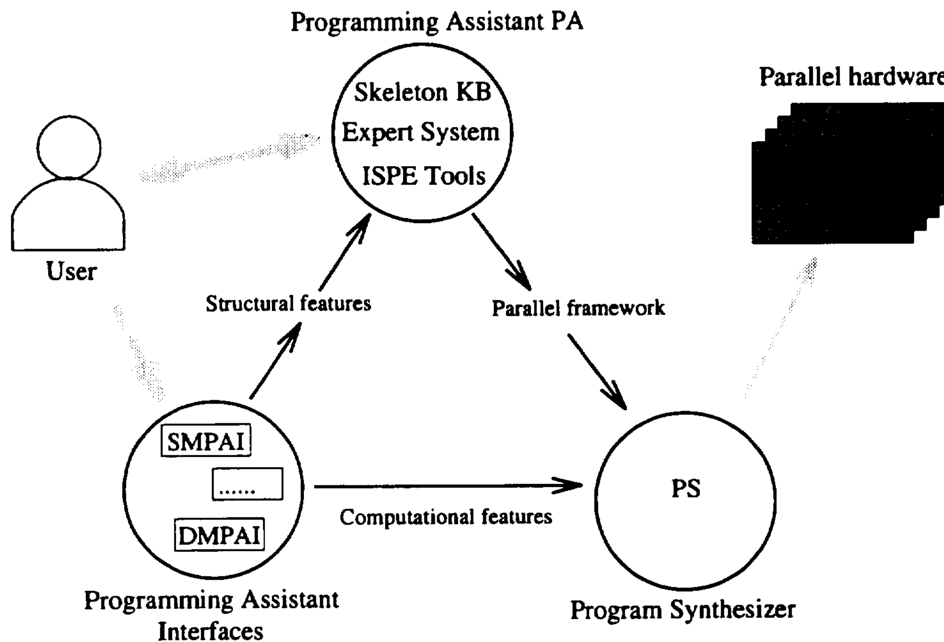

Figure 1: The conceptual structure of the PDE

session, the programmer gives an initial specification of the problem under consideration using a *programming assistant interface*. Currently, we have an interface *SMPAI* covering the domain of stencil-based applications on n-dimensional grids. Other interfaces are in preparation. The initial problem specification is decomposed into the purely computational features and the features relevant for the parallel structure. The first are passed directly to the *Program Synthesizer PS*, the latter go to the *Programming Assistant PA*. Then, in interaction with the user, the PA extracts and completes the information needed to determine an appropriate parallel framework. The PA is the central, largely AI-based component of the PDE, relying on various kinds of expert knowledge. The program synthesizer expands the parallel framework into compilable, hardware specific, parallel C++ or C programs.

3 REPRESENTATION OF ALGORITHMIC KNOWLEDGE

The PDE embodies a skeleton-oriented approach [Col89] to knowledge-based software engineering. A large collection of hierarchically organized fine-grain skeletons forms the algorithmic knowledge. The hierarchy spans from a general, abstract skeleton at the root to highly problem-specific, optimized skeletons at the leaf-level. Skeleton nodes in the tree represent a number of different entities. First, every non-leaf skeleton is a decision point for the rule-based descent. The descendants are specializations of the actual node in one particular aspect, the *discrimination criterion* of the skeleton. Every skeleton node contains information about the discrimination criterion and value that leads to its selection among the descendants of its parent node. The discrimination criteria represent concepts from parallel programming, application domain, and characteristics the supported parallel hardware architectures. Second, a skeleton node holds information about requirements other than the discrimination criterion that have to be checked before a descent is done. Third, skeleton nodes have slots for other kind of information that is not checked during descent but that may be helpful for the user or for subsequent steps. Finally, the skeleton nodes have attached methods or message handlers that are able to dynamically generate the parallel framework for the problem to be solved with the chosen skeleton.

Based on these different roles, the question arises whether the skeleton knowledge base should be built using instance or class objects. To access information in slots, a skeleton node should be an instance of a suitable class. However, to attach specialized methods or message-handlers at any node, skeletons have to be represented using classes. Also, the specialization type of the hierarchy suggests the use of a class hierarchy for the skeleton tree. We have solved the conflict by using both a class and an instance as the representation of a single skeleton node. A simplified example skeleton class is:

```
(defclass skel-32 (is-a init-bnd-mixin skel-5)
  (pattern-match reactive)
  (slot discrimination-crit (default problem-type) (create-accessor read))
  (slot discrimination-value (default INIT_BND_VAL_PROBLEM)
                             (create-accessor read))
  (multislot constraints (create-accessor read-write))
  (multislot required (default (create$ gridinstances_requirement))
                      (create-accessor read)))
```

The `required` slot holds instances of the requirements mentioned above. Constraints are

an additional concept used for describing restrictions in application scope.

All instances of the skeleton nodes are generated automatically with a simple function:

```
(deffunction create-instances (?rootclass)
  (progn$ (?node (class-subclasses ?rootclass inherit))
    (make-instance ?node of ?node)))
```

The two basic operations on the skeleton tree are the finding of the optimal skeleton, i.e., the descent in the tree, and the generation of the computational framework based on a selected node.

3.1 Rule-based skeleton tree descent

The descent in the skeleton tree is driven by rules. Based on the discrimination criterion of the current node, rules match on particular characteristics of the application specification and try to derive a value for the criterion. If a descendant node with the derived criterion value exists, it will get the candidate for the descent. The following example rule tries to find a descendant matching the grid coloring property of the specification:

```
(defrule DESC::coloring-1
  ?obj <- (object (name [descent_object])
                  (curr_skel ?skel))
  ?inst <- (object (discrimination-crit coloring)
                   (discrimination-value ?c))
  (test (member$ (class ?inst) (class-subclasses (class ?skel))))
  ?res <- (object (nr_of_colors ?c))
  ?app <- (object (is-a PAapp_class)
                  (results ?res))
  =>
  (send ?obj put-next_skel ?inst)
  (send ?obj put-decision_ok t))
```

The pattern matching relies heavily on object patterns. The whole expert system is programmed practically without the use of facts. The rule first matches a descent object, then a skeleton which is a descendant of the current skeleton and finally the `nr_of_colors` slot of the problem specification. Note that the particular ordering of object patterns is induced by some peculiarities of the CLIPS 6.0 pattern matching procedure. In order to handle references to objects in other modules, it is advisable to use instance addresses instead of instance names. CLIPS does not match objects from imported modules when they are refered to by a name property and the module name is not explicitly given. On the other hand, CLIPS has no instance-address property for object patterns and the form `?obj <- (object ...)` does not allow `?obj` to be a bound variable. So, the ordering of patterns is restricted in such a way that first an object gets matched and then it is verified whether it is the desired one. The consequence is probably some loss in pattern matching efficiency.

Although our initial problem domain is sufficiently restricted and well-defined to allow a complete, automatic descent in most cases, there is a collection of user interaction tools, called the *Intelligent Skeleton Programming Environment* ISPE, ready to handle the case

when the descent stops before reaching a sufficiently elaborated skeleton. With the ISPE, the user can add missing information, select a descendant node manually, or even step through the tree guided by the expert system. For such a functionality, the ISPE needs a high integration with the expert system. The rules for the descent are divided into various modules, separating search for descendants, verification and actual descent in different rule modules. The object of class `descent_class` used in the rule above serves to keep track of the current state of descent and to coordinate between the rule modules.

3.2 Generating output from the skeleton tree

After successful selection of a skeleton node for the application under consideration, the parallel framework can be generated. This is done by calling message-handlers attached to all skeletons. An example parallel framework for a simple stencil problem is:

```
init(Grid);
FOR (iter = 0; iter < nr_of_iterations; iter++) DO
 FOR (color = 0; color < 3; color++) DO
  INTERACTION
      fill_buf(Grid, w_obuf, west, color);
      Exchange(e_ibuf, east, w_obuf, west);
      scatter_buf(Grid, e_ibuf, east, color);
      .....
  END;
  CALCULATION
      update(Grid, color);
  END;
 ENDFOR;
ENDFOR;
finish(Grid);
```

The building blocks of this formalism [BG94] are communication and computation procedure calls, grouped by sequencing or iteration statements.

Object-oriented concepts are realized for optimizing the representational efficiency in the skeleton tree. First, *inheritance* insures that information stored in the slots of the skeleton nodes is passed down the tree. If from a certain point on it does not apply any more, it can be overriden. Second, new behaviour is introduced with *mixin classes*. As it can be seen in the `skel-32` node shown earlier, each node has both a mixin class and the parent class in its superclass list. In this way, it is possible to define a particular feature only once, but to add it at various points in the tree. Finally, instead of having methods that for each skeleton individually generate the complete parallel framework, an incremental *method combination* approach has been chosen. Every method first calls the same method of its parent node and then makes its own additions. The message-handler below shows this basic structure:

```
(defmessage-handler stencil-MS-mixin generate-MS ()
  (bind ?inst (call-next-handler))
  (send ?inst put-global_vars
       (create$ .... )))
```

Message-handlers instead of methods have to be used for this kind of incremental structure, as methods in CLIPS do not have dynamic precedence computation, whereas message-handlers do. Using methods, the sequence of skeleton and method definitions in the source file would be dominant for the precedence instead of the class hierarchy at runtime.

4 INTERFACES AND PROBLEM REPRESENTATIONS

The expert system of the PDE has basically three interfaces to the outside. It receives input from the problem specification tools, generates the parallel framework as output for the program synthesizer component, and it has an interface for user interaction.

4.1 Input

In order to omit a parser written in CLIPS, the formalism for the problem specification was chosen to be directly readable into CLIPS. A natural way to achieve this consists in using instance-generating constructs. We have two types of constructs in the input formalism, for example:

```
(make-instance global_spec of global_spec_class
               (grid_const 1.0)
               (dimension 2)
               (problem_type BND_VAL_PROBLEM))

(stencil_spec (assign_op (grid_var f (coord 0 0))
                         (function_call median
                                        (grid_var f (coord -1 0))
                                        (grid_var f (coord 0 0))
                                        (grid_var f (coord 1 0)))))
```

In the first statement above, where global properties of the application are defined, the `make-instance` is part of the formalism. Obviously, reading such a construct from a file with the CLIPS `batch` command generates an instance of the respective class and initializes the slots. In the second case, where the stencil is defined, the instance-generating ability is not directly visible. However, for every keyword such as `stencil_spec`, `assign_op` or `grid_var`, there are functions creating instances of respective classes, e.g.:

```
(deffunction grid_var (?name ?coord)
  (bind ?inst (make-instance (gensym*) of grid_var))
  (send ?inst put-var_name ?name)
  (send ?inst put-coord ?coord))
```

The reasons to not use `make-instance` in all cases are that some constructs can have multiple entities of the same name, e.g., a stencil can have multiple assignment operations, and additional processing that is needed by some constructs.

The lack of nested lists in CLIPS was considered as a disadvantage at the beginning, in particular regarding the close relation between CLIPS and Lisp/CLOS. However, a recursive list construct can be easily defined with COOL objects and instance-generation

functions similar to the one above. Certainly, this is not appropriate if runtime efficiency is critical. In the course of the PDE development, the lack of lists proved to have a positive effect on style and readability. For example, instead of using just the list (-1 1 2) for a 3-D coordinate, using the construct (coord -1 1 2) creates an instance of a specialized class for coordinate tuples. Such an instance can be easier handled than a list and appropriate methods or message-handlers can be defined. The whole problem representation after reading the specification input is present in a number of nested, interconnected instances of respective classes.

4.2 Output

The result of the generation of the parallel framework is a problem representation by means of a number of instances, much in the sense of the input representation shown above. The writing of an output file as shown in section 3.2 is done with message-handlers attached to each of the relevant problem representation classes, e.g., a for-loop is written by:

```
(defmessage-handler for_loop_class write-out (?stream)
  (printout ?stream "  FOR (" ?self:varname " = " ?self:from
                    "; " ?self:varname " < " ?self:till
                    "; " ?self:varname "++) DO " t)
  (progn$ (?el ?self:subs)
          (send ?el write-out ?stream))
  (printout ?stream "  ENDFOR;" t))
```

5 EMBEDDING WITH C++

The global structure of the PDE consists of components written in C++, among them the main program and the graphical user interface, and an embedded CLIPS expert system for knowledge representation and reasoning. Additionally, two parsers use the Lex/Yacc utilities. Whereas some components, such as the parsers, are integrated only by means of intermediate files for input and output, the expert system is highly integrated with the C++-based ISPE and the graphical user interface. The CLIPS dialog itself is visible to the user through a CLIPS base window. Figure 2 shows both the CLIPS dialog window and a browser window for graphically browsing the skeleton tree.

5.1 Data integration

Both CLIPS and C++ offer objects for the representation of data. It is therefore a straightforward decision to use the object mechanism for the data integration between an expert system written in CLIPS and programs written in C++. The concept for the CLIPS-C++ integration relies on the decisions to represent common data on the CLIPS side using COOL objects and to provide wrapper classes on the C++ side for a transparent access to COOL objects. A class hierarchy has been built in C++ to represent CLIPS types, including classes and instances. Access to COOL classes is needed for example in the skeleton tree browser, where descendants of a node are only found by inspecting the subclasses list of the node.

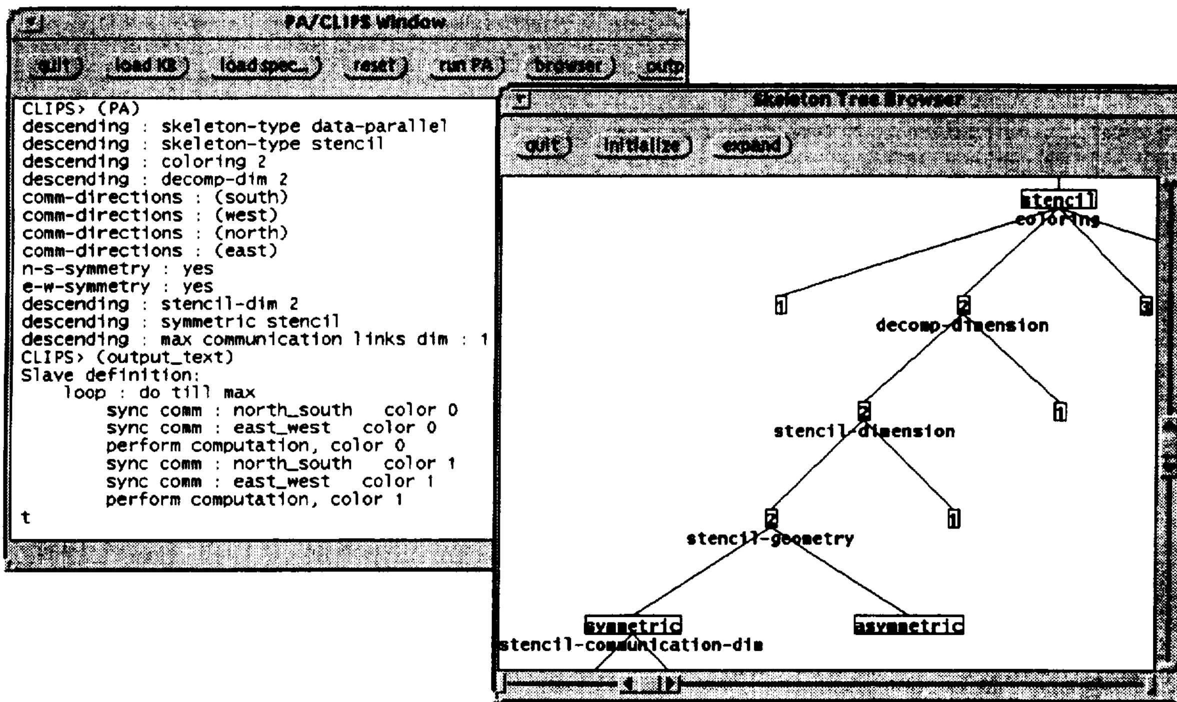

Figure 2: The CLIPS window and the skeleton tree browser

The C++ wrapper classes consist of an abstract class `clips_type` with subclasses for the CLIPS data types integer, float, symbol, string, multifield, and for COOL classes and instances. The class `coolframe` used to represent COOL instances is shown below:

```
class coolframe : public clips_type {
  char* framename;
 public:
  coolframe(char* name);
  clips_type* get_slot(char* slotname);
  int put_slot(char* slotname, clips_type* value);
  char* class_of(); };
```

The creation of a C++ frontend to a COOL instance is performed by just instantiating the above class, giving the constructor the name of the COOL instance. Accesses to slots have to be done by using the `get_slot` and `put_slot` member functions that refer to functions in the CLIPS external interface [NAS93]. The class `coolframe` is a generic class, usable for any COOL instance, no matter what collection of slots the COOL instance has. A more sophisticated approach with separate members for all slots would be somewhat more convenient, as the distinction between accessing a C++ object and accessing a COOL object through the C++ wrapper would be completely blurred. However, defining classes on the fly, based on the structure of the COOL instances, is not possible in C++. The chosen system with general-purpose wrapper classes is already convenient for use in C++. Operators and functions in C++ can be overloaded to handle CLIPS data transparently. For example, the basic mathematical operations can be overloaded to combine C++ numbers with CLIPS numbers in one expression.

Care has to be taken to not introduce inconsistencies between the data stored in CLIPS and the C++ wrappers. To this end, the C++ interface to CLIPS does not cache any

information, and before performing any access through the CLIPS external interface it is verified whether the COOL object to be accessed still exists.

5.2 Functional integration

The goal with functional integration is to achieve fine-grain control over the reasoning in the expert system from C++-based components whenever needed. For example, it is sometimes desirable to check whether one single descent from the current skeleton node is possible. Or, the user may prefer to step manually through the tree, getting support from the expert system. The detailed control needed for such tasks has been achieved with a partitioning of the rules into a number of rule modules. Then, the C++ component can set the focus to just the rule system desired and start the reasoning.

Two basic means to interact from the C++ components to the CLIPS expert system exist in the PDE. First, a C++ program can call any of the C functions from the CLIPS external interface. This happens without user visibility in the CLIPS dialog window, and a return value can be obtained from the call. Second, thanks to the graphical user interface written in C++, the C++ program can directly write to the dialog window in such a way that CLIPS thinks it received input at the command line. Using this alternative, no return value can be passed back to the C++ component, but the interaction is visible to the user in the window. It is thus most suited to starting the reasoning or other functions that produce output or a trace in the dialog window.

6 CONCLUSIONS

The realization of the parallel program development environment PDE has successfully achieved the primary goal of making parallel programming as simple as sequential programming within the initial problem domain of stencil-based applications. Moreover, thanks to programming environment support spanning from the design level up to automated code generation and thanks to the reuse of important software components, parallel programming with the PDE can be considered substantially simpler and more reliable than sequential programming in a common procedural language. Apart from the high-level, domain-oriented approach to programming, the PDE offers to the user efficiency preserving portability of software across platforms, reuse of critical software components, and a flexible and comfortable interface.

With a focus on the parts realized using CLIPS, various aspects of the implementation of our knowledge-based parallel program development tool PDE have been shown in this paper. CLIPS in its current version 6.0 [NAS93] proved to have some critical properties making it particularly well-suited for the use within the PDE. Of highest importance are probably the object-oriented capabilities of CLIPS, enabling flexible interfaces to the outside, appropriate representations of knowledge and intermediate problem states, and, together with the CLIPS external interface, a convenient embedding with the C++ components of the PDE. CLIPS is well suited for a rapid prototyping approach to system development, in particular due to the flexibility that the object-oriented mechanism can offer. The PDE development is currently in the fourth prototype. Apart from the first throw-away prototype done in CLOS [BDG+88], each prototype reuses large parts of the previous one, adding completely new components or new functionality.

The problems with the CLIPS language encountered during the PDE development relate in part to the resemblance of CLIPS to CLOS. Examples are the lack of the lists in the sense of Lisp, the static precedence determination for methods, or the inability to pass methods or functions as first-class objects. But alternatives offering similar functionality have been found in all cases. Apart from such CLOS-like items, a suggestion for improvement of the CLIPS language based on our experience is to focus more on object patterns in rules than on facts or templates. An useful extension of the CLIPS external interface, based on the popularity of the C++ programming language, would be the definition and documentation of a C++ frontend for COOL objects.

REFERENCES

[BDG+88] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. *Common Lisp Object System Specification.* X3J13 Document 88-002R, 1988.

[BG94] H. Burkhart and S. Gutzwiller. Steps towards reusability and portability in parallel programming. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 147 - 157. Birkhäuser Verlag, Basel, 1994.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA., 1989.

[DDR94] K.M. Decker, J.J. Dvorak, and R.M. Rehmann. A knowledge-based scientific parallel programming environment. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 127 - 138. Birkhäuser Verlag, Basel, 1994.

[DR92] K.M. Decker and R.M. Rehmann. Simple and efficient programming of parallel distributed systems for computational scientists. Technical Report IAM-92-019, IAM, University of Berne, 1992.

[Dvo93] J. Dvorak. An AI-based approach to massively parallel programming. Technical Report CSCS-TR-93-04, Swiss Scientific Computing Center CSCS, CH-6928 Manno, Switzerland, 1993.

[GR94] J. Giarratano and G. Riley. *Expert Systems: Principles and Programming.* PWS Publishing, Boston, MA., 2nd. edition, 1994.

[NAS93] NASA Lyndon B. Johnson Space Center. *CLIPS Reference Manual*, 6.0 edition, 1993.

[Pol88] C.D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, 1988.

[ZC90] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers.* Addison-Wesley, Workingham, 1990.