Program Comprehension Engines for Automatic Parallelization: A Comparative Study

Beniamino Di Martino* Christoph W. Keßler†

Abstract

We compare two systems for program comprehension that are targeted towards support of automatic parallelization: the PAP recognizer currently included into the Vienna Fortran Compilation System, and the PARAMAT pattern recognizer developed at Saarbrücken University. We illuminate the main differences, the advantages and disadvantages of each approach, and show how both approaches may be integrated to combine the generality of one approach with the speed of the other one.

1 Introduction

Program comprehension is the process of discovering abstract concepts in the source code. In its generality, it does not seem automatable; but, given some knowledge on the program's application domain, automatic understanding is possible at least on a local level. In this case, understanding becomes a recognition process: it can be sketched as matching a given source program against a set of known programming patterns.

A number of problems have to be dealt with when facing automated recognition of algorithmic patterns [16]. The most important ones are *syntactic variation*, *algorithmic variation* (a pattern can be implemented in many different ways), *delocalization* (the implementation of a pattern may be spread throughout the code), and *overlapping implementations* (portions of a program may be part of more than one pattern instance).

Automatic program comprehension systems are mainly used for two purposes: to support software maintenance, e.g. for automatic documentation of code, and to support automatically parallelizing compilers. Several methods for both areas have been proposed within the last years, and some (mostly experimental) systems have been built. Here we focus on the second issue.

Automated program recognition can play a crucial role in overcoming limitations of existing tools for automatic parallelization for distributed-memory architectures. For instance, replacement of recognized sequential code by suitable parallel algorithms overcomes an important limitation of existing automatic parallelizers that are able to parallelize some loops but are still bound to the sequential program's control structure. Moreover, the acquired knowledge enables automatic selection of sequences of optimizing transformations, supports automatic array alignment and distribution, and improves accuracy and speed of performance prediction [8].

*Inst. for Software Technology and Parallel Systems, University of Vienna, Austria, and DIS - University of Naples, Italy. email: `dimartin@par.univie.ac.at`

†FB 4 Informatik, University of Trier, D-54286 Trier, Germany. email: `kessler@psi.uni-trier.de`

The application domain considered mainly consists of numerical computations, in particular linear algebra and PDE codes. Domain analysis [8] has shown that the size of the set of patterns typically occurring in such codes remains reasonably small. But also in non-numerical or irregular numerical fields, recognition of algorithmic patterns in the code can drive the selection of suitable parallelization strategies, as [4, 5] shows for the *Divide-and-Conquer* pattern in Quicksort implementations and the *Branch-and-Bound* pattern in optimization codes.

We present two automatic program comprehension engines that have mainly the same goals but vary considerably in their methods, properties, and implementations. We compare these approaches in detail, showing their advantages and disadvantages, relate them to earlier work and conclude with a proposition to combine the speed of one method with the flexibility of the other one.

2 Deterministic Program Recognition in PARAMAT

PARAMAT's pattern recognizer works on the intermediate representation of the source program as an abstract syntax tree. A well–structured and statically analyzable source language is assumed. The goal is to annotate as many nodes as possible with a so–called *pattern instance*, a summary structure that describes which function is computed in the subtree rooted at that node, together with the parameter objects of that function. Speed and robustness of this method mainly result from exploiting the natural semantic hierarchy of the patterns in the library.

Preprocessing First we apply several normalizing transformations to make the program as explicit as possible, by inlining all procedures (recursive procedures are very untypical for the application area considered), forward propagation of constant expressions, recognition and replacement of induction variables, and eliminating dead code.

Semantic hierarchy of patterns Each pattern consists of a specification of a (mathematical) operation and of the types and the data structures of its parameters.

For instance, the MV pattern represents the operation $\vec{y} = \mathbf{A}\vec{b}+\vec{x}$, with the parameters $\vec{y}$, $\mathbf{A}$, $\vec{b}$, and $\vec{x}$ being real (sub-)arrays ($\vec{x}$ may also be a constant).

For each nontrivial pattern, we usually know by experience many implementation prototypes (for sequential C code). Because of the wide variety of semantics preserving code transformations, the number of such prototypes can be tremendous for more complex patterns (such as matrix–matrix multiplication), blowing up the size of an automatically generated tree automaton dramatically. For this reason, we formulate the prototypes as far as possible by using instances of (other) patterns. E.g., an implementation of matrix–vector product (MV) can be written as a single loop around a dot product

```
for (i=1; i<=n; i++)
   SSP( x[i], A[i][1:m], b[1:m], x[i]);
```

or as a loop summing up the result vectors of vector triads

```
for (j=1; j<=m; j++)
   VAADDSV( x[1:n], b[j], A[1:n][j], x[1:n]);
```

because $(\mathbf{A}\vec{b}+\vec{x})_{i=[1:n]} = (\sum_{j=1}^{m} A_{ij}b_j + x_i)_{i=[1:n]} = \sum_{j=1}^{m}((A_{ij}b_j)_{i=[1:n]})_j + (x_i)_{i=[1:n]}$.

With such domain information it becomes straightforward to formulate *templates*, that are the rules to determine a node's pattern m (and pattern instance I) given the node's operator and all its children's pattern (i.e., subpattern) instances. In the case of several children, we select for each template the most characteristic child, and call the expected pattern for this child the *trigger pattern*. For instance, there is a template for MV with trigger pattern SSP, and another one with trigger pattern VAADDSV. For each pattern, we realize only the most important templates matching it (typically, we have 1 to 3 realized templates per pattern), see [8] for the complete list.

This natural semantic pattern hierarchy is stored in a directed graph (*pattern hierarchy graph, PHG*) where the patterns are the nodes and for each template an edge goes from the trigger pattern to the pattern to be inferred. Thus pattern recognition becomes a path finding problem in the PHG. Besides from cycles from a pattern to itself, the PHG is acyclic. Different paths towards a pattern m correspond to different implementations of the functionality of m. Thus, a linear–sized PHG (and thus, pattern recognizer) represents exponentially many implementation variations of the same pattern.

The PHG has a second important advantage: it serves also as a hash table that can be inspected by the pattern recognition algorithm, as it yields immediately all possible superpatterns that could be matched from a given trigger pattern. Often, the trigger pattern together with the operator of the node to be matched suffices to select a single possible template. If there are several templates admissible, these are tested concurrently; the result is deterministic. Failing templates abort as soon as possible.

Recognition algorithm The abstract syntax tree is traversed from left to right in postorder. For a leaf node (a variable or a constant), it is trivial to determine its pattern (VAR or CONST, respectively). At each inner node v of the syntax tree, the algorithm tests, based on v's operator and v's children's patterns already matched, whether there is a pattern m in the library (there exists at most one) which matches the semantics of the subtree T_v rooted at v. Selection of admissible templates is done by inspecting the PHG. For each of them a short routine is called that realizes that template; it fails if it cannot prove that the function computed by T_v equals the operation represented by m, and returns an instance I of pattern m otherwise. In the latter case, it also maps the program objects to the slots of I (pattern parameters), and annotates v with I.

Before trying to match a new loop header, the algorithm distributes [17] that loop as far as possible and applies pattern matching to each of the resulting loop headers separately. Loop distribution is often supported by scalar expansion [17], using temporary arrays.

The basic method is extended for pattern matching along 'horizontal' dataflow edges, such that several (matched) instructions in the same block that belong to the same pattern may be contracted to a single pattern instance, even if they are textually separated. Several instructions may belong to the same thread of computation only if their operands are involved in at least one of several types of dataflow relations that we denote by dataflow edges. These *cross edges* in the syntax tree represent particular, loop-independent data flow relations among the operands of pattern instances within the same block. Thus they are well–suited to guide 'horizontal' pattern recognition [9]. Computing exact array data flow is generally hard, but here we can profit from the simple array access structures that are characteristic for dense matrix computations and that are present in all our patterns. Matching along cross edges is particularly helpful to disentangle intermixed computations, thus guiding pattern recognition, or to reroll unrolled loops.

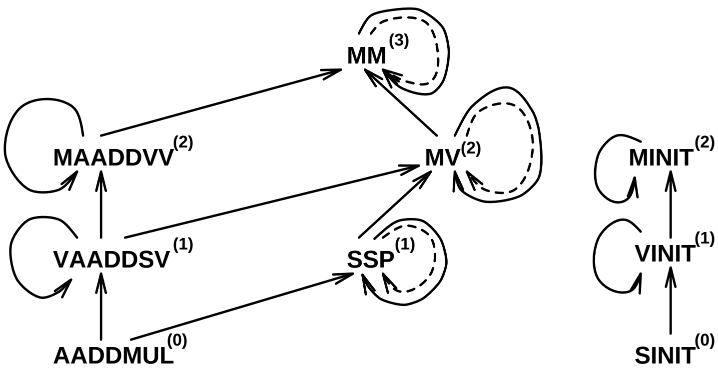
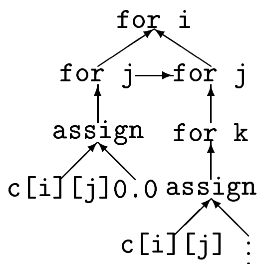


Figure 1: The pattern hierarchy graph of Matrix–matrix multiplication. Solid edges mean realized templates for 'vertical' pattern recognition; dashed ones for 'horizontal' pattern recognition along cross edges. Solid cycles mean templates for unblocking or elimination of semantically invariant conditionals; dashed cycles represent templates for loop rerolling or integration of initializers.

Example We demonstrate the pattern recognition algorithm at a simple example. Matrix–matrix multiplication is well suited since the functionality of it and of its subpatterns is widely known, and since its PHG (Figure 1) is rather clear. Suppose the following program fragment is given:

```
     for (i=1; i<=n; i++) {
       for (j=1; j<=m; j++)
S1:      c[i][j] = 0.0;
       for (j=1; j<=m; j++)
         for (k=1; k<=r; k++)
S2:        c[i][j] = c[i][j]+a[i][k]*b[k][j];
     }
```



The algorithm traverses the abstract syntax tree from left to right in postorder. First, it encounters at `S1` a scalar initialization `SINIT (c[i][j], 0.0)`. For the `j` loop around it, we obtain an instance of a vector initialization `VINIT( c[i][1:m], 0.0)`. The access to array `c` has become a vector as one dimension has been bound by the loop.

Next, assignment `S2` is considered and annotated by `AADDMUL (c[i][j], a[i][k], b[k][j], c[i][j])` (accumulative addition of a product). Following the suitable PHG edge, this yields a dot product for the `k` loop: `SSP (c[i][j], a[i][1:r], b[1:r][j], c[i][j])`. The accesses to the arrays `a` and `b` have become vectors. As the accumulating scalar `c[i][j]` has not been initialized so far, it has to be entered into the initialization slot of the SSP instance to keep data access information consistent. Then, the `do j` loop around the SSP instance is recognized as an instance of matrix–vector multiplication. Again the accumulating vector `c[i][1:m]` fills the initialization slot. The partially matched, unparsed syntax tree now looks as follows (code parts 'below' recognized nodes are not shown any more):

```
for (i=1; i<=n; i++) {
  VINIT(c[i][1:m], 0.0);
  MV(c[i][1:m],a[i][1:r],b[1:r][1:m],c[i][1:m]);
}
```

At this point we need data flow information in order to continue recognition. Here we obtain that vector `c[i][1:m]` is written in the VINIT instance, and read and overwritten by the MV instance. A *cross edge* of type FLOW symbolizes that data flows between these two instances in an expected way. This situation can be tested by a realization of another template for pattern recognition along cross edges. As the template matches, these two instances are merged into a single MV instance `MV (b[1:r][1:m], a[i][1:r], 0.0)`, i.e., the initialization slot is now filled by `0.0` from the VINIT instance. This instance, in turn, can be matched with the `i` loop into `MM(c[1:n][1:m], a[1:n][1:r], b[1:r][1:m], 0.0)` (matrix–matrix multiplication) representing this entire piece of code.

During pattern recognition, we have followed the PHG paths SINIT...VINIT and AADDMUL...SSP...MV...MV...MM. Common program transformations, like loop interchange or loop distribution, would result in a different path being taken towards MM, but would not prohibit pattern recognition.

Postprocessing Finally we must eliminate useless code that may emanate from conservative cross matching and certain transformations. Instances of *unstable patterns* are decomposed into their basic patterns' instances. E.g., general vector operations are decomposed into simple vector operations using temporary arrays.

Implementation The current prototype implementation consists of 12000 lines of C code and reliably recognizes 91 nontrivial patterns with 150 nontrivial templates. Each template realization is implemented as a C routine of 20 to 50 lines. Since many useful syntactic and semantic predicates have been predefined, writing code for templates is handy and straightforward. More patterns can easily be added. Robustness against loop interchange, loop distribution, loop unrolling and statement reordering has been exemplified in practice as well as the high speed of the recognition algorithm [8].

3 Backtracked program recognition: the PAP Recognizer

PAP (*Parallelizable Algorithmic Patterns*) Recognizer is a (prototype) tool for automated program comprehension, aimed towards support of code parallelization. It implements a *plan based* technique for the recognition of *concept instances* in the code, that works in a hierarchical way. It provides as output a graphical browser that visualizes the hierarchy of recognized concepts and their position in the source code. It is mainly driven by the semantic features (control, data dependence, calling relationships) of the concepts. This, combined with the backtracking feature of the recognition procedure, offers high flexibility that allows to handle non-numerical and irregular codes as well as numerical ones.

The prototype relies on Prolog as system shell, thus taking advantage of its deductive inference engine. It utilizes the structural analysis of the input code performed by the *Vienna Fortran Compilation System* [18] (VFCS) front-end, an interactive compilation system for distributed memory parallel computers, in which PAP Recognizer has been integrated as parallelization support tool [6].

Recognition process PAP Recognizer performs a hierarchical parsing process, driven by *concept recognition rules*, that acts on *concept instances* descriptions.

The matching rules of the hierarchical concept parsing (*plans*) are production rules that describe the features of the concepts to be recognized. Features identifying an (algorithmic) concept can be informally defined as the way some abstract functions (the composing subconcepts) are related and organized into a specific abstract control structure. By "abstract control structure" we mean structural relationships, such as control flow, data flow and calling relationships. These relationships involve "abstract" objects, i.e. aggregates of variables or statements linked by a functionality.

More specifically, each concept is recursively specified by its compositional hierarchy, and by relationships and constraints among composing subconcepts.

With regard to the control and data dependence relationships, to put in evidence their peculiarity and to simplify the concepts specification, they are subjected to an abstraction process during the recognition, likewise the concept abstraction. Indeed, a notion of *abstract* control and data dependency between abstract concepts has been introduced, and defined in a recursive way: a concept instance C_i is defined as *abstract data (control) dependent* on another concept instance C_j, if the composing subconcepts of C_i satisfy a particular pattern of abstract data (control) dependence relationships with the concept C_j; this pattern is characteristic of each concept C_i, and is determined by the plan which recognizes it.

The terminals of the recursive specification of abstract concepts, and abstract data and control dependence relationships among them, are represented by the set of syntactical, control and data dependences on the program, obtained by a structural analysis of it, and codified as *base* concepts and dependence relationships.

A top-down direction (demand driven recognition) has been chosen for the hierarchical parsing. This is due to the particular aim of the recognition process, that is trying to find instances of *Parallelizable Algorithimc Patterns* in the code (algorithmic patterns for which a (set of) parallelization strategies can be defined, related to the underlying architecture). For this purpose, a recognition of the functionalities of the whole program is not needed, and thus a bottom-up parsing is not necessary. If the list of high level PAPs to be searched in the code is well defined (especially in relation to the characteristics of the underlying architecture), and if the recognition plans are cleverly designed in such a way to fail as soon as possible, the top-down approach allows for a deep pruning of the search space associated with the hierarchical parsing.

Abstract program representation The output of the structural analysis phase, that is syntactic, control flow, data flow and data dependence informations on the program, is utilized to build the *Base Internal Representation* of the program, which is stored in the *Concept Instances Database* in the form of *base concepts*. This base level of the representation is substantially a Program Dependence Graph (PDG), whose nodes represent statements and whose edges represent control and data dependences. This slightly differs from a standard PDG graph, in that it is augmented with the following structures:

- control dependence edges are labeled with the branch (true, false) of the dependence, and data dependence edges are labeled with the dependence variable identifier and the kind of the dependence (loop independent or loop carried);
- assignment statement nodes are augmented with two tree structures representing its left and right hand sides; subscript expressions of array variable instances are likewise represented by tree structures, linked to the node representing the array variable instance;
- each control statement node is augmented with the tree structure representing the control conditional expression;
- each node of the augmented PDG points to the corresponding nodes of the syntax tree, so that a direct reference from the abstract concept recognized to the code implementing it is possible; in this respect, the PDG could be viewed as a kind of web;
- variable definitions are explicitly represented, including information about the type, the rank for array variables, and including pointers to the corresponding nodes of the

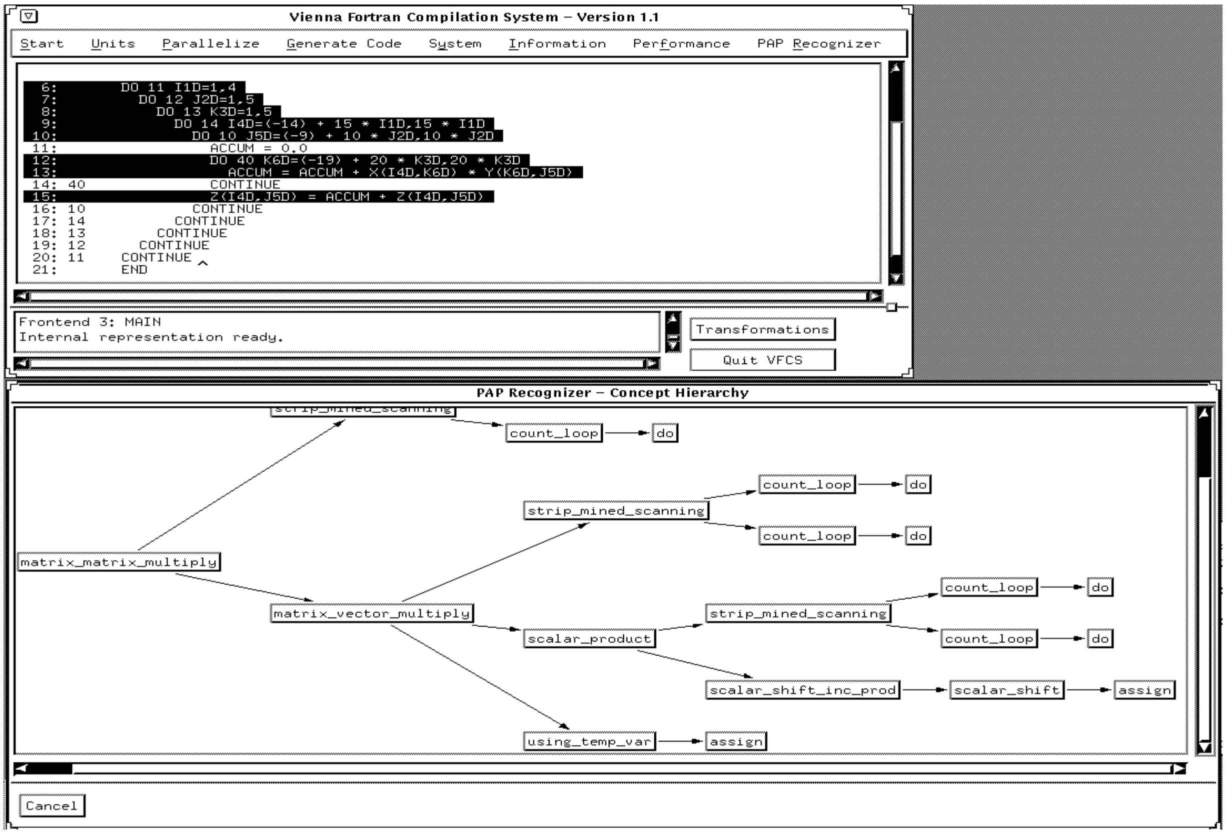


Figure 2: Recognition of a tiled implementation of matrix–matrix multiplication

syntax tree.

The overall internal program representation is generated during the recognition phase performed by the PAP Recognizer. It has the structure of a *Hierarchical PDG* (HPDG), reflecting the hierarchical strategy of the recognition process. Starting from the base internal representation, the recognizer performs the concept parsing, following the rules specified by the plans. 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. More specifically, each node representing a concept is linked to the nodes of the lower layer representing subconcepts. Abstract control and data dependence edges for the newly created abstract concept nodes are inherited from those of the composing subconcept nodes, in a way that is characteristic for each concept, and which is determined by the plan which recognizes it. Internal Program Representation is stored in the Concept Instances Database which is incrementally updated by the PAP recognizer as the recognition advances.

Features The syntactic variation problem is solved by: (1) characterizing inter–statement level concepts with non-syntactic properties like control and data dependence that are a better characterization of concepts than other lower level features; (2) taking advantage of the backtracking characteristic of the recognition procedure to perform symbolic analysis of expressions within statements. The delocalization problem is solved by the characteristics of the abstract program representation which: (1) is based on an inherently delocalized structural representation (PDG); (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. 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 plans: each plan specifies a different algorithmic implementation of the same concept. 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 plan, 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 this approach does not need a canonical form for concept implementations.

Example We show how an instance of the matrix–matrix multiplication algorithmic pattern, implemented with tiling of the iterations, is recognized in a code fragment. Fig. 2 (upper window) shows the Fortran program segment normalized by the VFCS front end. This version is analyzed by the PAP Recognizer, and Fig. 2 (lower window) shows the hierarchical structure of the recognized pattern, in the form of a graph. By clicking on each of the graph nodes, representing a composing subconcept, it is possible to highlight the corresponding code implementing it, as shown in the figure for the overall concept instance. As can be seen in this figure, the tool is able to recognize the concept in the presence of a temporary accumulator variable for the scalar-product subconcept; the implementation variation represented by the tiling is recognized by composing the matrix multiplication concept with the concepts of strip mining and loop interchange, as can be seen in the figure representing the hierarchical structure.

4 Detailed Comparison

Determinism Both approaches are hierarchical. But there is a decisive difference:

The PARAMAT recognizer is deterministic: each syntax tree node can carry at most one pattern instance which summarizes anything that is contained in the subtree below it. This is made possible by applying loop distribution. Moreover, the PARAMAT recognition is leveled: Because at each level there is only a very limited number of candidates that may match the node, the speed of recognition is very high.

The PAP Recognizer applies backtracking: this allows for dealing with implementation variations and delocalization of concept instances. Moreover, the concept matching is not limited in the level of abstraction: the particular structure of the abstract program representation (the hierarchical abstract PDG) make it possible to have a global scope of visibility of the recognized subconcepts, thus allowing the recognition even in presence of sharing of concept instances. These two features make recognition more powerful, but also make the search complexity grow exponentially with the code size. Nevertheless, both the top down approach, that permits to not inspect all the code, and the summarization of derived subconcept within HPDG nodes, prune the search space considerably.

Program representation and representation of recognized features The PARAMAT recognizer uses the abstract syntax tree of the source program as intermediate representation. Data flow information is computed incrementally during the recognition

system	PAP Recognizer	PARAMAT pattern recognizer
direction	top–down	bottom–up
driving features	syntax and semantic driven (control, data dependence)	syntax driven (data flow used to deal with delocalization)
backtracking	yes, essential for the method	not necessary (deterministic)
restructuring	not necessary for the method	essential (canonicalization)
preprocessing	not necessary	necessary
power of recognition	more powerful (sharing, delocalization, variation of implementations)	less powerful
speed	slow	fast
goals	alignment and distributions selection; replacement by library routines only if coupled with restructuring; template based par. code generation	alignment and distributions sel., replacement by library routines

Table 1: Main differences between the two approaches

process and only where needed. If required, the syntax tree can be transformed during the recognition process. — A recognized code portion, here always identical to a subtree of the abstract syntax tree, is summarized in a pattern instance which annotates the root of the matched subtree. The pattern instance contains all the information of what is computed in that subtree (but not any more *how* it is computed). A pattern instance I annotating a node v can be used for two purposes: (1) It can be used to determine the pattern of the father of v. There is no need of I for the recognition process any more after v's father has been also matched. (2) It offers the possibility of code replacement.

The PAP Recognizer starts using the PDG as intermediate representation. During the recognition process, it builds upon the PDG an abstract hierarchical PDG that represents the hierarchy of recognized concepts. Although associated with specific program locations and program objects, this abstract representation is not limited in its scope of visibility but globally visible, thus allowing for subconcept sharing and recognition of delocalized concepts. Each derived subconcept is summarized in its functionality, compositional hierarchy and linking to the syntax tree nodes (and thus to the code segments implementing it) by its node in the HPDG.

Recognition process The PARAMAT recognizer is driven by the syntax tree structure and by the data flow edges computed meanwhile. Each instance of a recognized pattern is one-to-one linked to a node in the abstract syntax tree. This makes pattern recognition deterministic and relates recognition to a traversal of the abstract syntax tree.

The PAP recognizer is driven by control and data dependence relationships of the (inter-statement level) concepts. The parsing process is not related to a syntax tree traversal; it is rule based and with backtracking feature. Rules are applied on a globally visible abstract program representation. 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 complexity can be maintained at an acceptable level, without constraining the delocalized recognition capability.

Restructuring during recognition PARAMAT's recognizer relies on powerful restructuring transformations such as loop distribution or scalar expansion. This simplifies loop bodies and allows to apply a simpler pattern matching algorithm. The modifications of the source program are motivated by the application domain (replacement by parallel routines) and are compatible with the code replacement algorithm [10]. The most important transformation in this context is loop distribution because it allows to factor out different computations from the same loop nest that can be matched separately by different patterns (and thus result in separate code being generated).

The PAP Recognizer doesn't need to restructure during recognition because it is not based on canonicalization of concept implementations. Syntactic variations (as textual nesting) are instead as much as possible made transparent to the recognition process by using non-syntactic features like control and data dependence. For the same reason, the approach doesn't need (if not to speed up the recognition) preprocessing transformations, and can thus deal with situations where these cannot be applied.

Sharing The PARAMAT recognizer restructures the program and applies loop distribution to factor out the shared code portions as far as possible. It then decides (if not directed by data dependences, by using a heuristic) the order of pattern instances and assigns shared code portions to the first of them. The intermediate results of the shared code portions used by subsequent pattern instances must be written to temporary variables. This is necessary because the goal is code replacement. Simple duplication of common code is not provided; indeed that would lead to incorrect code in some cases.

The PAP recognizer has no problems with code portions shared by several patterns, because of the nonlevelled and nonlocalized matching mechanism, and because of the possibility to reuse derived subconcepts for other matchings.

5 Related Work

Several automatic program comprehension techniques have been developed over the last years. They vary considerably in their application domain, method, and status of implementation.

Earlier work targeted towards automatic code optimization or parallelization Snyder [15] addresses idiom recognition in APL codes. His algorithm is an extended depth–first traversal of the abstract syntax tree with linear expected run time. He applies dynamic programming techniques to select the most profitable idiom in the presence of overlapping idioms, which appears to be common in APL programs. — [3] suggests (non–constructively) to apply pattern matching techniques for the detection of reductions and recurrences within the framework of a formal system for automatic shared memory parallelization. — EAVE [2] is an expert system for interactive vectorization of FORTRAN programs. It contains a simple pattern matching tool that can decover order 1 patterns (vector operations, reductions). — The pattern matcher of [12] works on a modified program dependence graph that has been extended in a special way to match certain loop structures with the goal of replacing them by parallel algorithms. The cost of recognition is higher because the rewrite rules form a graph grammar. Normalization of the PDG has to be provided interactively by the user. — [13] proposes a special approach for recurrence detection. While this method offers, at considerable computational effort, the recognition of rather general and multidimensional recurrences, a number of assumptions are made that are hardly met by real applications. As complicated recurrences are rare in real programs, the computational effort of this approach seems not justified. — CMAX [14] is the only commercial application of pattern matching with regard to parallelization. It translates FORTRAN77 programs to CM-Fortran, a parallel vendor–specific Fortran dialect similar to Fortran90. It recognizes syntactically several common loop constructs (vector operations, reductions, matrix–matrix multiply) but does not distinguish between patterns and templates. The recognition power is slightly weaker than PARAMAT's, but the main advantage of CMAX is its ability to recognize FORTRAN-specific storage conventions and to transform them to make the program machine-independent and more suitable to distribution of data in that point. — Similar to the systems compared in this paper is the first phase of a program comprehension system for FORTRAN programs sketched in [11]. It works top–down on the PDG and partly uses the algorithm from [15].

Other problem domains Some systems for program comprehension in a non–numerical domain are targeted towards automatic documentation and support of software maintenance. Plan Calculus [16] represents code and patterns (called "clichés") with graph structures whose nodes correspond to subconcept instances and whose arcs capture control and data flow relationships among them. Clichés recognition becomes thus a *graph parsing* process using a set of *graph grammar rules*. It produces a parse tree representing a hierarchical description of plausible concepts of the program. — PAT [7] uses an abstract, object-oriented representation for syntactic and semantic concepts composing the program. Each concept is an instance of a concept class, and the classes are hierarchically structured. Our templates are roughly comparable to their "plans"; a plan's representation consists of a description of the syntactical components and a description of the constraints to be satisfied by components. An inferential pattern–directed engine derives new higher–level concepts from the existing ones, utilizing plans as inference rules.

6 Conclusion

We have presented two systems for automatic program comprehension, the PARAMAT pattern recognition tool and the PAP recognizer. Both systems are targeted towards comprehension of numerical codes (even though the flexibility of the second approach makes it suitable to deal with irregular or non-numerical code too) with the goal to support automatic parallelization.

Nevertheless, the systems vary considerably in their methods, properties, and implementations. While, roughly speaking, PARAMAT's pattern recognizer provides acceptable (for the given domain) recognition power at impressive speed and its output is suited for code replacement more straightforwardly, the PAP recognizer offers more flexibility and generality at the expense of higher run time. Thus, there is just a tradeoff between power of recognition and speed to be deliberated before choosing the one or the other system for a certain application program, depending on its size and complexity.

We are currently studying the possibility and effectiveness to combine both approaches. One possibility could be the following. If recognition time is very critical, we recommend to use PARAMAT's pattern recognizer for longer codes. For shorter codes, or if recognition time is less critical, the greater flexibility of the PAP recognizer should be exploited. If pure PAP recognition takes too much time and one is willing to trade recognition power for time, we propose that the PARAMAT pattern recognizer could be run as a preprocessor; the pattern instances produced become then facts for the PAP recognizer. Restructuring done by the PARAMAT recognizer is finished before the PDG for PAP recognition is constructed. The already recognized program parts are not submitted to PAP recognition any more, resulting in a downsized PDG. This corresponds to explicit a priori pruning of the PAP search tree and clearly limits the flexibility of the PAP recognizer. The degree of preprocessing could be modified by the user, e.g. by limiting PARAMAT's recognition to some restricted set of patterns. It is up to the user to decide about that when recognition time is critical.

References

[1] S. Bhansali, J.R. Hagemeister, C.S. Raghavendra, H. Sivaraman, "Parallelizing sequential programs by Algorithm-level Transformations", 3^{rd} *IEEE Workshop on Program Comprehension*, Washington, Nov. 1994.

[2] P. Bose. Interactive Program Improvement via EAVE: An Expert Adviser for Vectorization. *Int. Conf. on Supercomputing*, pp 119–130, July 1988.

[3] T. Brandes, M. Sommer. A Knowledge-Based Parallelization Tool in a Programming Environment. *Int. Conf. on Parallel Processing*, pp 446–448, 1987.

[4] B. Di Martino, G. Iannello. Towards Automatic Parallelization through Program Comprehension. 3^{rd} *IEEE Workshop on Program Comprehension*, Washington, Nov. 1994.

[5] B. Di Martino, B. Chapman. Program Comprehension Techniques to improve Automatic Parallelization. *Workshop on Automatic Data Layout and Performance Prediction*, Rice University, Houston TX, Apr. 1995.

[6] B. Di Martino, B. Chapman, G. Iannello, H. Zima. Integration of Program Comprehension Techniques into the Vienna Fortran Compilation System. *Int. Conf. on High Performance Computing*, New Delhi (India), Dec. 1995.

[7] M. Harandi, J. Ning. Knowledge-Based Program Analysis. *IEEE Software*, Jan. 1990.

[8] C.W. Keßler. *Automatische Parallelisierung numerischer Programme durch Mustererkennung.* PhD thesis, Universität Saarbrücken, 1994.

[9] C.W. Keßler. Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. *IFIP WG10.3 Working Conf. on Progr. Environments for Massively Par. Distr. Systems.* Birkhäuser, Apr. 1994.

[10] C.W. Keßler. Pattern-Driven Automatic Program Transformation and Parallelization. *3rd EUROMICRO Workshop on Parallel and Distributed Processing, San Remo.* Jan. 1995.

[11] R. Metzger. Automated Recognition of Parallel Algorithms in Scientific Applications. *Workshop on Plan Recognition at IJCAI'95*, Aug. 1995.

[12] S.S. Pinter, R.Y. Pinter. Program Optimization and Parallelization Using Idioms. *ACM SIGPLAN Principles of Programming Languages*, pp 79–92, 1991.

[13] X. Redon, P. Feautrier. Detection of Recurrences in Sequential Programs with Loops. *PARLE 93, Springer LNCS vol. 694*, pp 132–145, 1993.

[14] G. Sabot, S. Wholey. Cmax: a Fortran Translator for the Connection Machine System. *Int. ACM Conference on Supercomputing*, pp 147–156, 1993.

[15] L. Snyder. Recognition and Selection of Idioms for Code Optimization. *Acta Informatica*, 17:327–348, 1982.

[16] L.M. Wills. Automated Program Recognition: a Feasibility Demonstration. *Artificial Intelligence*, **45**, 1990.

[17] H. Zima, B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison–Wesley, 1990.

[18] H. Zima, B. Chapman. Compiling for Distributed-Memory Systems. Proc. of the IEEE, Special Section on Languages and Compilers for Parallel Machines, Feb. 1993.