UNIVERSITÄT TRIER
Mathematik / Informatik

Forschungsbericht Nr. 97-23

Two Program Comprehension Tools for Automatic Parallelization: A Comparative Study

Beniamino Di Martino Christoph W. Keßler

Electronic copies of technical reports are available:

Via FTP: Host ftp.informatik.uni-trier.de, directory /pub/reports

Via gopher: Host gopher.informatik.uni-trier.de, port 70, type 1, pathes 1/TechReports/Abstracts and 1/TechReports/FullText

Via WWW: URL http://www.informatik.uni-trier.de/Reports/List

Via email: Send a mail to ftpmail@ftp.informatik.uni-trier.de, subject 'MAIL ME CLEAR', body 'TechReports.HowTo' followed by an empty line, for detailed instructions

Printed copies:

Trierer Forschungsberichte
Fachbereich IV -
Mathematik / Informatik
Universität Trier
D-54286 Trier

ISSN 0944-0488

Two Program Comprehension Tools 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 concept recognizer developed at Saarbrücken University. We illuminate the main differences, the advantages and disadvantages of each method, and show how both approaches may be integrated to combine the generality of one approach with the speed of the other one.

Key words: automatic program comprehension tools, program concept recognition, automatic parallelization, automatic code restructuring, parallel library reuse.

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 of 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 concepts.

A number of problems have to be dealt with when facing automated recognition of algorithmic concepts [7]. The most important ones are *syntactic variation*, *algorithmic variation* (a concept can be implemented in many different ways), *delocalization* (the implementation of a concept may be spread throughout the code), and *overlapping implementations* (portions of a program may be part of more than one concept 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.

[‡]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`

Automated program recognition can play an important role in enhancing the capabilities of existing parallelizing compilers for distributed-memory architectures. For instance, replacement of recognized sequential code by suitable parallel algorithms represents an aggressive code transformation which is not 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 (see the sidebar in Figure 2 which summarizes the application areas of automatic program comprehension with respect to program parallelization).

The application domain considered mainly consists of numerical computations, in particular linear algebra and PDE codes (see the sidebar in Figure 1). Domain analysis [5] has shown that the size of the set of concepts typically occurring in such codes remains reasonably small. But also in non-numerical or irregular numerical fields, recognition of algorithmic concepts in the code can drive the selection of suitable parallelization strategies, as [1] shows for the *Divide-and-Conquer* concept in Quicksort implementations and the *Branch-and-Bound* concept in optimization codes.

We present two automatic program comprehension tools that have mainly the same goals but vary considerably in their methods, properties, and implementations. between We describe the functionality of the two methods using the well–known matrix–matrix multiplication as an example (more concepts they can handle are listed in the sidebar in Figure 1). We identify common features and differences between these approaches, show 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 Hierarchical Concept Recognition

A *concept* consists of a specification of a (numerical) operation and of the types and the data structures of its parameters.

For instance, the MV concept represents the matrix–vector multiplication $\vec{y} = \mathbf{A}\vec{b} + \vec{x}$, with the parameters $\vec{y}$, $\mathbf{A}$, $\vec{b}$, and $\vec{x}$ being real (sub-)arrays.

The goal is to annotate a region of code with a summary node, called a *concept instance*, which represents the operation covered by this code and contains the program objects occurring as the operation's parameters. For instance, given a simple matrix–vector multiplication C code

```
for (i=1; i<=n; i++)
  for (j=1; j<=m; j++)
    x[i] = x[i] + A[i][j] * b[j];
```

we would like to come up with a concept instance like

```
MV( x[1:n], A[1:n][1:m], b[1:m], x[1:n]);
```

Figure 1: **Some examples of concepts in scientific programs**

The computational cores of numerical application programs operating on large arrays representing vectors and dense matrices are typically composed of a quite limited set of program concepts. The following list is far from being exhaustive but should give an impression of some concepts that are of interest for automatic parallelization, together with simple example occurrences (note that our tools can handle much more complicated cases):

Basic scalar operations:	
numerical quadrature	`RES=INIT; DO I=2,N-1 RES=RES+FUNC(I)`
difference stencil (1D, 2D)	`B(I,J)=Q(I,J)+A(I,J)+.25*(A(I-1,J)` `  +A(I+1,J)+A(I,J-1)+A(I,J+1))`
swap, Givens rotation	`TMP=C*A+S*B; B=C*B+S*A; A=TMP`
Basic linear algebra operations:	
vector / matrix sum / max / min	`RES=INIT; DO I=1,N RES=RES+A(I)`
elementwise vector / matrix operations	`DO I=1,N-1 DO J=1,M C(I,J)=A(J,I)+B(I+1)`
dot product	`DOT=INIT; DO I=1,N DOT=DOT+A(I)*B(K,I)`
prefix sum / first order lin. recurrence	`DO I=2,N A(I)=B(I)+C(I)*A(I-1)`
matrix-vector product	`DO I=1,N DO J=1,M C(I)=C(I)+A(I,J)*B(J)`
matrix-matrix product	`DO I=1,N DO J=1,M DO K=1,L` `  C(I,J)=C(I,J)+A(I,K)*B(J,K)`
Solving linear equations and PDE systems:	

LU, QR decomposition	`DO I=1,N { A(I,I)=1.0/A(I,I);`
	`  DO J=I+1,N { A(J,I)=A(J,I)*A(I,I);`
	`    DO K=I+1,N A(J,K)=A(J,K)-A(J,I)*A(I,K)}}`

Jacobi / JOR relaxation (1D, 2D)	`DO I=2,N-1 DO J=2,M-1`
	`  B(I,J)=Q(I,J)+A(I,J)+.25*(A(I-1,J)`
	`    +A(I+1,J)+A(I,J-1)+A(I,J+1));`
	`DO I=2,N-1 DO J=2,M-1 A(I,J)=B(I,J)`

Gauß–Seidel / SOR relaxation (1D, 2D)	`DO I=2,N-1 DO J=2,M-1`
	`  A(I,J)=A(I,J)+.25*(L(I,J)*A(I-1,J)`
	`    +R(I)*A(I+1,J)+D(J)*A(I,J-1)+U*A(I,J+1));`

Image processing:	
Convolution (1D, 2D)	`DO I=1,N DO J=1,M DO K=1,I DO L=1,J` `  Y(I,J)=Y(I,J)+U(K,L)*V(I-K,J-L)`
Fast Fourier Transform	*(e.g., recursive divide-and-conquer implementation)*
Sorting and searching:	
min / max location (1D, 2D)	`K=1;X=A(1); DO I=2,N IF (A(I)>X) {K=I;X=A(I)}`
sorting an array	*(e.g., recursive quicksort implementation)* [a]

[a] see B. Di Martino, G. Iannello: Towards Automatic Parallelization through Program Comprehension. In V. Rajlich and A. Cimitile, editors, *Proceedings of the* 3^{rd} *IEEE Workshop on Program Comprehension*, Washington, Nov. 1994. pp. 108–115. Los Alamitos: IEEE Computer Society Press.

Figure 1: **Some examples of concepts in scientific programs**
(continued)

For instance, in the Livermore Loops[b], a benchmark suite consisting of 24 representative computational kernels (LL1...LL24) taken from scientific application programs, there occur concepts like dot product (LL3), first–order linear recurrence (LL5, LL11), 1D minimum location (LL24), matrix–matrix multiplication (LL21), 1D Jacobi relaxation (LL8) and 2D Gauß–Seidel relaxation (LL23). 16 of the 24 kernels can be (mostly completely) covered by the concepts from a catalog[c] containing 150 frequently occurring basic numerical concepts. The set of routines typically offered by (parallel) numerical library packages is quite similar to this catalog.

[b] see J. T. Feo: An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computing* **7**, pp. 163–185, 1988.

[c] see C. W. Keßler: Pattern–Driven Automatic Parallelization. *Scientific Programming* **5** pp. 251–274, 1996.

Figure 2: **Applications of Automatic Program Comprehension in Parallelization**

Automatic program restructuring / library reuse. The derived knowledge on the semantics of the recognized code parts may guide sophisticated program transformations up to the replacement of a recognized code fragment by a call to an optimized parallel library routine implementing the same functionality.

Automatic data distribution. When compiling for distributed–memory machines, computing optimal alignment and distribution schemes for the arrays in the program is crucial for the performance but, unfortunately, an NP–complete global optimization problem. While array distribution induces work distribution in a straightforward way, badly chosen array distributions, may cause excessive data communication or even sequentialize computations. If combined with replacement by calls to parallel library routines, the information provided by our concept instances can be used to look up solutions that are locally optimal (for the parallel library routine) as starting points for the solution of the global problem. Furthermore, the routines may be parameterized in array distributions, and after solving the global problem, customized versions could be generated by partial evaluation. [a]

Automatic performance prediction. Solving the global array distribution problem means finding a combination of distribution schemes which minimizes the (estimated) run time of the program. Program comprehension enhances run time estimation by treating concept instances as units, i.e. the run time function of the concept implementation is looked up from precomputed run time tables, instead of being computed bottom–up.

[a] see C. W. Keßler: Pattern–Driven Automatic Parallelization. *Scientific Programming* **5** pp. 251–274, 1996.

Figure 2: **Applications of Automatic Program Comprehension in Parallelization** (continued)

Speculative program restructuring. An interesting point are sparse matrix computations: as the space–efficient data structures used there rely on run–time data, static program comprehension has not sufficient information available at compile time. Speculative program comprehension[e] combines compile–time concept recognition with suitable run–time checks that are to be inserted into the speculatively generated code where necessary.

Independence from a particular parallel execution model. Another important point is the current limitation of parallelizing compiler technology, represented by the use of a fixed parallel execution model, even a very flexible and general one like the SPMD model. It may prevent the selection of a parallelization strategy that is best suited to the characteristics of the code to be parallelized. Several authors[f g h] have argued that parallel programs can be classified as belonging to a *Parallel Programming Paradigm* (also called *template* or *skeleton*), based on the way processes forming the parallel program are created, synchronize and communicate, while abstracting from the details of their sequential part. Automatic program comprehension can play an important role in overcoming this limitation of parallelizing compiler technology: it can drive the automatic selection of the parallel execution model that is best suited to the algorithm and to the target architecture, and derive the final parallel code, by applying *template based* code transformations[i].

Knowledge-based support to interactive parallelization. The information gathered by the PAP Recognizer is used[j] in the Vienna Fortran Compilation System[k] together with knowledge-based and expert systems techniques to guide the user in his/her interactive decisions for loop parallelization, code restructuring, and array alignment and distribution.

[e] see C. W. Keßler: *Applicability of Program Comprehension to Sparse Matrix Computations.* Proceedings of Euro-Par'97, Passau, August 1997, Springer Lecture Notes in Computer Science.

[f] see: P. Brinch Hansen: Model programs for computational science: a programming methodology for multicomputers, *Concurrency: Practice and Experience*, 5(5), pp. 407–423, Aug. 1993.

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

[h] see: I. Foster and M. Xu: "Libraries for Parallel Paradigm Integration", in *Toward Teraflop Computing and New Grand Challenge Applications*, Nova Science Publishers, 1994.

[i] see B. Di Martino, G. Iannello. Towards Automatic Parallelization through Program Comprehension. In V. Rajlich and A. Cimitile, editors, *Proceedings of the* 3^{rd} *IEEE Workshop on Program Comprehension*, Washington, Nov. 1994. IEEE Computer Society Press, Los Alamitos, Calif., pp. 108–115.

[j] see S. Andel, B. Di Martino, I. Hulman, H. Zima, "Program Comprehension Support for Knowledge Based Parallelization", Proc. of 4^{th} *EUROMICRO Workshop on Parallel and Distributed Processing*, Braga (PT), IEEE, Jan. 1996.

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

covering this piece of code. The more complex computations can be covered by concept instances, the more accurate information becomes available for code generation.

For each nontrivial concept, we usually know by experience many *idioms*, that are parameterized implementation prototypes for this concept in a given programming language like C or Fortran77. Because of the wide variety of semantics preserving code transformations, the number of idioms can be tremendous for more complex concepts such as matrix–matrix multiplication, see the example code in Figure 8 for illustration. Thus, providing a separate recognition rule for each idiom would require too much space and search time. For this reason, we formulate the idioms as far as possible in terms of instances of (other) concepts. E.g., a C 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 $(\mathbf{A}\vec{b}+\vec{x})_{i=[1:n]} = (\sum_{j=1}^{m} A_{ij}b_j + x_i)_{i=[1:n]} = \sum_{j=1}^{m}((A_{ij}b_j)_{i=[1:n]})_j + (x_i)_{i=[1:n]}$. In this example, the dot product concept SSP and the vector multiply–add concept VAADDSV appear as *subconcepts* in idioms for MV. The subconcept relation installs a "natural" semantic hierarchy of concepts. Exploiting this hierarchy is crucial for efficient concept recognition.

With such domain information it becomes straightforward to formulate the *rules* to infer a concept and, if successful, to compute an instance of the concept. A rule is specified in terms of intermediate–representation *features* which characterize the concept to be recognized. Such features may be operator symbols, equality of program objects, control and data flow information, and already computed subconcept instances. To make this work in practice, instances of concepts already recognized must be stored in the intermediate program representation the concept recognizer is working on, as well as the other features. This extended intermediate representation may be e.g. an extended program dependence graph or an annotated abstract syntax tree. In the former case, a graph–parsing method is required; the latter case allows application of customized tree pattern matching techniques.

Some aspects of the intermediate representation can depend on the particular goals the method could be devoted to.

One possible goal is to derive sophisticated information for the user that gives him/her advice for *interactive parallelization*. In this case, the source program (and thus also the intermediate representation) should be read–only since the user has to decide about application of transformations and parallelization. Many possibilities for transformation, data layout, parallelization etc. may seem worth consideration by the parallelizer, but generally not all of them can be exploited since some of them will conflict with each other.

A different goal is to have a program comprehension engine tailored towards *automatic replacement* of program parts (as large as possible) by calls to (given) optimized parallel library routines. In this case the tool is expected to *restructure* the intermediate representation in a *deterministic* way such that the concept instances directly translate to subroutine calls. This restructuring can either be applied during the recognition process, which implies that the tool implicitly has some knowledge about the parallel implementations and anticipates some decisions from the parallelization phase, or a separate restructuring step is applied afterwards [3].

In either case, the subsequent compiler phases are only interested in instances of *stable concepts*, i.e. concepts for which a (set of) parallelization strategies can be defined, related to the underlying architecture, or, respectively, for which a parallel routine exists.

3 Deterministic Program Recognition in PARAMAT

PARAMAT [5] is a (prototype) system which tries to replace recognized code parts by calls to parallel library routines. PARAMAT's concept 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 concept instance, here a summary structure that describes which function is computed in the complete *subtree* rooted at that node, together with the parameter objects of that function. Speed and robustness of this method mainly result from exploiting the hierarchy of the concepts in the library.

Preprocessing First we apply several normalizing transformations [8] 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.

Exploiting the semantic hierarchy of concepts For each concept, we realize only the most important rules matching it; typically, we have 1 to 3 realized rules per concept.

In order to speed up finding an applicable rule, we exploit the hierarchy of concepts. Generally, applicable rules at each syntax tree node considered are triggered by already recognized subconcepts and the node's operator. For each rule we select the most characteristic subconcept occurring in it, called the *trigger concept*. The trigger concept relation is explicitly stored in a directed graph (*"pattern hierarchy graph", PHG*) where the concepts are the nodes, and for each realized rule, an edge goes from the trigger concept to the concept to be inferred. Thus concept recognition becomes a path finding problem in the PHG. Besides from cycles from a concept to itself, the PHG is acyclic. Different paths towards a concept c correspond to different implementations of the functionality of c. Thus, a linear–sized PHG (and thus, concept recognizer) represents exponentially many idioms of the same concept.

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

Recognition process 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 concept (VAR or CONST, respectively). At each inner node v of the syntax tree, the algorithm tests, based on the operator and the children's concepts already recognized, whether there is a concept c in the library (there exists at most one) which matches the semantics of the subtree T_v rooted at v. Selection of admissible rules is done by inspecting the PHG. For each of them a short routine is called that realizes that rule. This routine fails if it cannot prove that the function computed by T_v equals the operation represented by c. Otherwise it annotates v with an instance of the recognized concept c and maps the program objects to the corresponding parameter slots.

Before trying to match a new loop header, the algorithm distributes [8] that loop as far as possible and applies concept recognition to each of the resulting loop headers separately. Loop distribution is often supported by scalar expansion [8], 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 concept may be contracted to a single concept 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 concept instances within the same block. Thus they are well–suited to guide 'horizontal' concept recognition [4]. 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 concepts. Matching along cross edges is particularly helpful to disentangle intermixed computations, thus guiding concept recognition, or to reroll unrolled loops.

Figure 4 illustrates how the algorithm works.

Postprocessing Finally we must eliminate useless code that may emanate from conservative cross matching and certain transformations. Furthermore, instances of non–stable

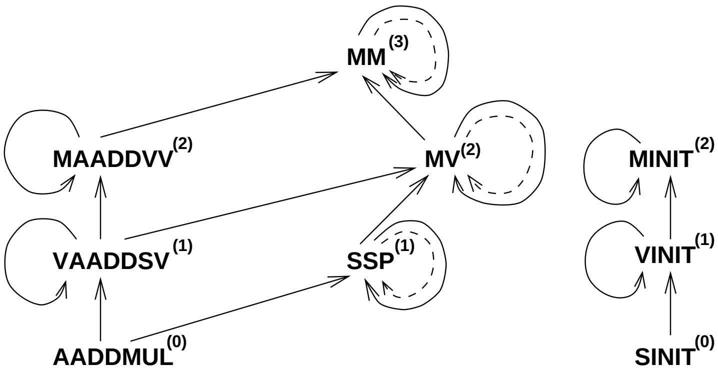


Figure 3: The concept hierarchy graph of Matrix–matrix multiplication. Solid edges mean realized rules for 'vertical' concept recognition; dashed ones for 'horizontal' concept recognition along cross edges. Solid cycles mean rules for unblocking or elimination of semantically invariant conditionals; dashed cycles represent rules for loop rerolling or integration of initializers.

concepts are decomposed into their basic concepts' instances. E.g., general vector operations are broken off into simple vector operations using temporary arrays.

Implementation The prototype implementation of the PARAMAT concept recognizer consists of 12000 lines of C code and reliably recognizes 91 nontrivial concepts with 150 nontrivial rules. Each rule 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 rules is handy and straightforward. More concepts 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 [5].

PARAMAT has a simple back end driver that emits Fork95 [6] source code for the shared–memory multiprocessor SB-PRAM where for each concept instance a call to a corresponding Fork95 library routine is generated.

4 Backtracked program recognition: the PAP Recognizer

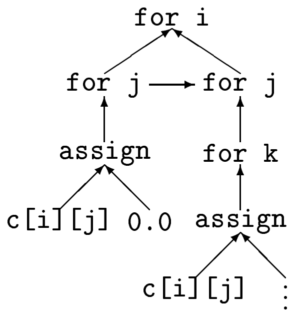
PAP Recognizer[1] [2] is a (prototype) tool for automated program comprehension, aimed towards automatic restructuring and parallelization, and computer-aided program analysis and parallelization. It currently provides as output a graphical browser that visualizes the hierarchy of recognized concepts and the code portions which represent their instances in the source code. Provision of features for automatic translation from Fortran 77/90 code to HPF, such as code replacement with HPF intrinsic calls, insertion of HPF directives for data and work distribution and application of sequences of unimodular loop transformations are currently under development [3].

[1] PAP stands for *Parallelizable Algorithmic Patterns* which is another term for the stable concepts.

Figure 4: **Example for concept recognition in PARAMAT**

We demonstrate the concept recognition algorithm at a Matrix–matrix multiplication C code. The PHG of the Matrix–Matrix–Multiplication concept MM and its most important subconcepts is given in Figure 3.

```
  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 rule for concept recognition along cross edges. As the rule matches, these two instances are merged into a single MV instance `MV (c[i][1:m], 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 concept 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 concept recognition.

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 to the approach.

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* [9] (VFCS) front-end, an interactive compilation system for distributed memory parallel computers, in which PAP Recognizer has been integrated as parallelization support tool. Figure 5 shows a block scheme representing the interaction among the components of the PAP Recognizer, and among them and the VFCS components.

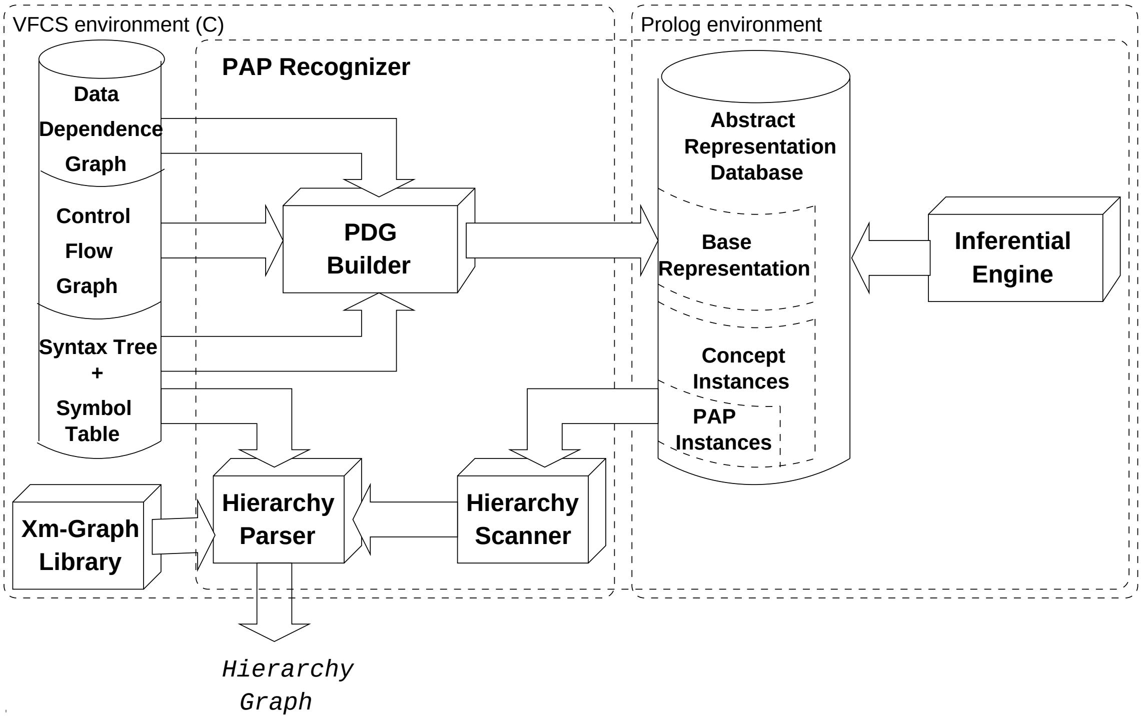


Figure 5: Scheme of the interaction among the PAP Recognizer's components, and among them and the VFCS components.

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 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 rule 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 the *stable* concepts. 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 concepts to be searched in the code is well defined (especially in relation to the characteristics of the underlying architecture), and if the recognition rules 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* (see Figure 5) 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, 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 is 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 pro-

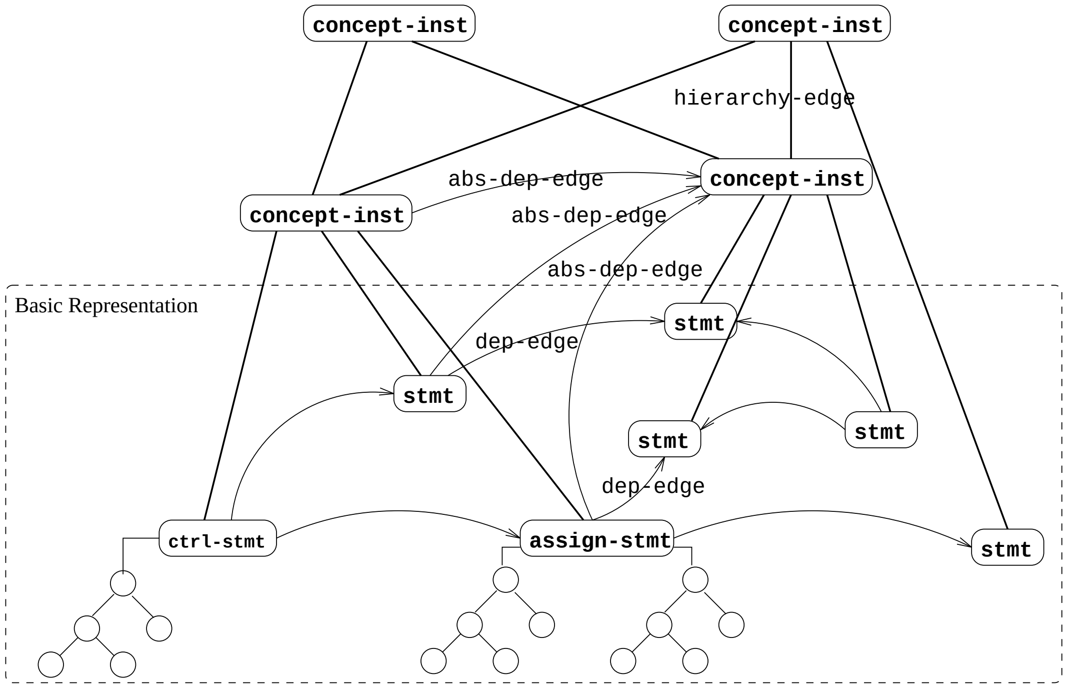


Figure 6: Abstract program representation.

cess on top of the basic representation (see Figure 6). 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, the concept instances are represented as nodes in increasingly higher layers of the HPDG. The nodes are connected by two kinds 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 for the case of shared concepts, i.e. when a concept instance is a subconcept of more than one concept (see Figure 6). The *abstract dependence edges* link together nodes that have abstract control and data dependence relationships between them. Note that, during the recognition process, abstract dependence edges for the newly created abstract concept instance nodes are inherited from those of the composing subconcept nodes in a way that is characteristic of each concept (see the annotated edges in Figure 6).

Features of the approach 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, and (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), and (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 rules: each rule 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 rule, allowing the recognition even in presence of shared concept instances.

An important consequence of these features is that restructuring of the original code before and during the recognition process is not required to deal with delocalized code and implementation variations. This is possible because this approach does not need to generate a canonical form for recognized code, which is left for a subsequent parallelization stage of the VFCS.

Example In Figure 7 and Figure 8 we show two examples for the recognition of the matrix-matrix multiplication concept (and, consequently, the composing subconcepts matrix-vector product and dot product).

The screen display visualizes the hierarchy graph. Clicking on each of the graph nodes, representing a (sub)concept instance, highlights the corresponding code implementing it.

5 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 concept 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 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 allows for a partial inspection of the code, and the summarization

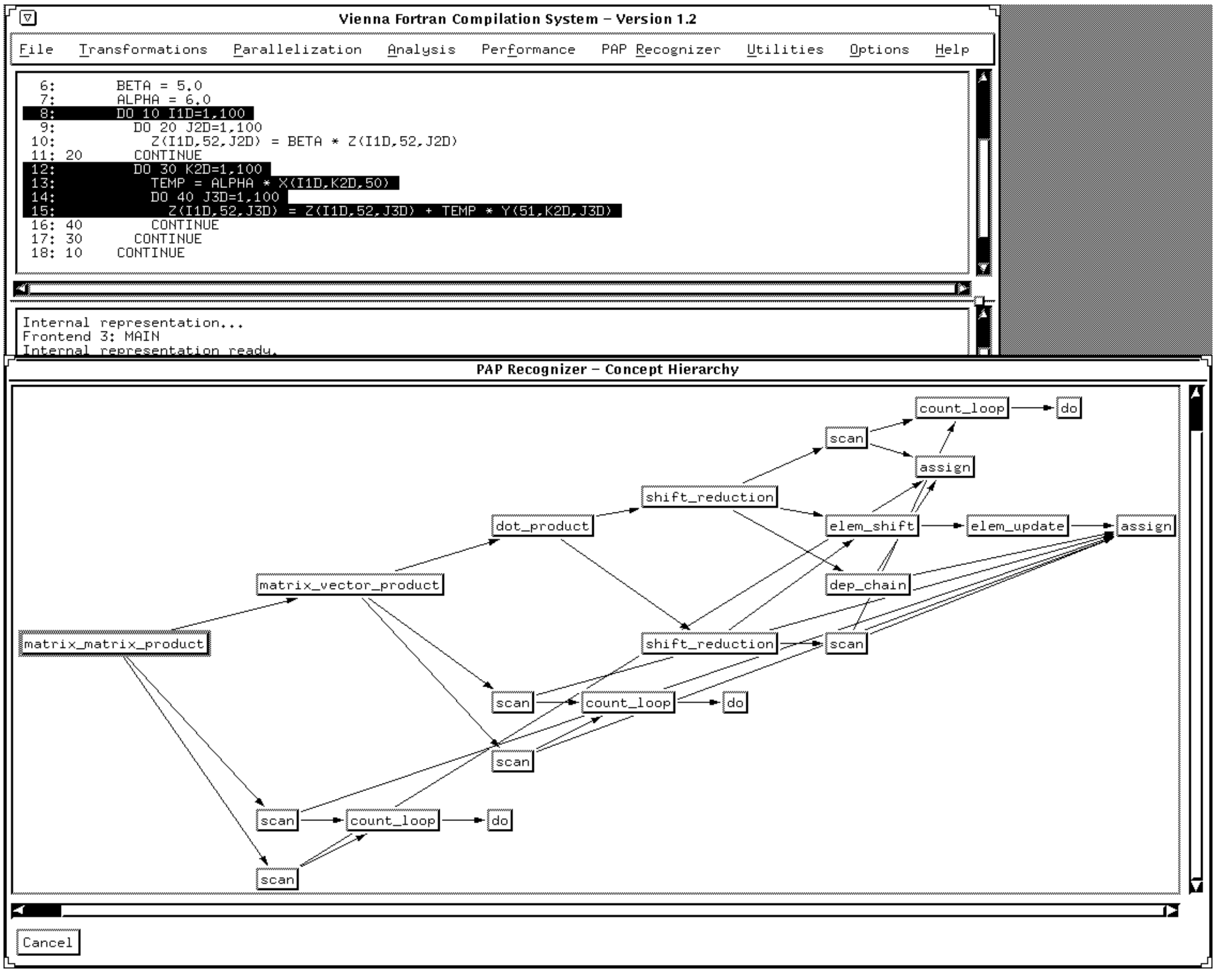
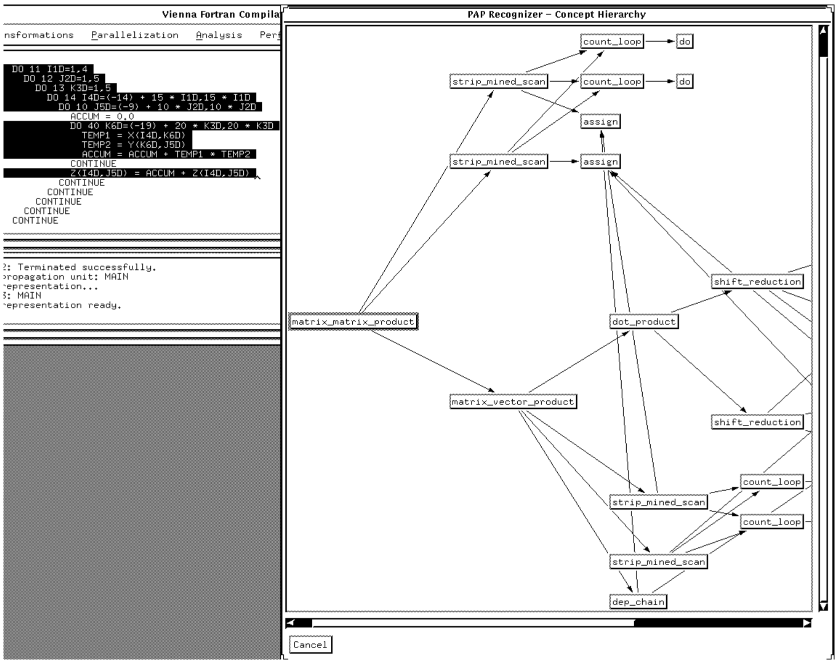
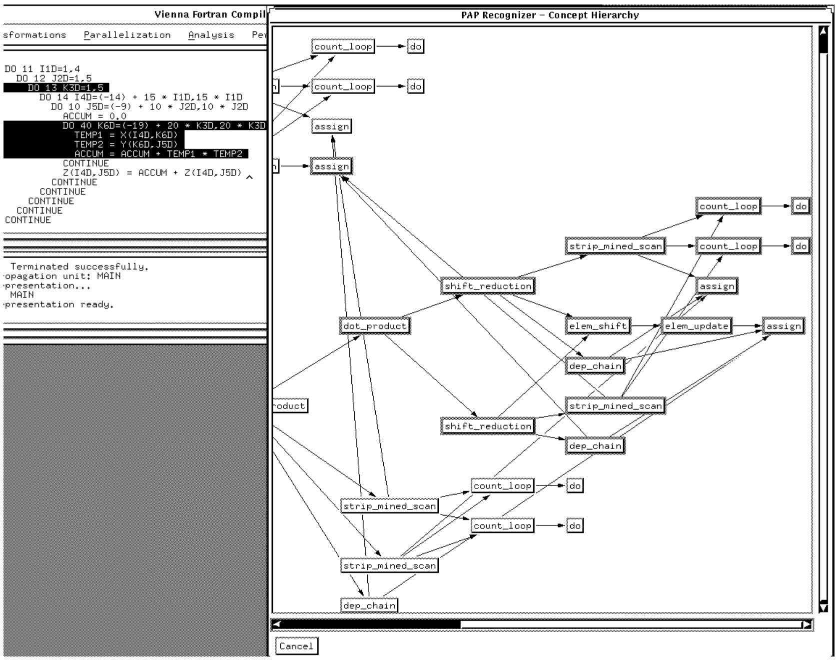


Figure 7: **First example for recognition of `matrix_matrix_product` by the PAP recognizer** This Fortran77 code excerpt is directly taken from the BLAS 3 implementation of the matrix-matrix product. The product is between arrays `X` and `Y`, both threedimensional ones. It involves the first and second dimension of `X` (the second involved in the dot product), and the second and the third of `Y` (the second involved in the dot product). The result is recorded in a page of the array `Z`, along the first and third dimension, whose values are updated after being scaled by `ALPHA`. The values of the page of `X` are also scaled by `BETA`, before the dot product operation. These operation are intermixed with the product implementation, but this does not represent a problem for the recognition. Moreover, the code excerpt presents an optimization, constituted by (1) a loop interchange between the loop involved in the dot product (line 12) and the one involved in the matrix vector product (line 14), and (2) the introduction of the temporary variable `TEMP`. This implementation variation is easily dealt by the recognition technique: the loops are properly recognized as belonging to their own subconcepts, and the use of the temporary variable is represented by a dependence chain (`dep_chain`).



(a) matrix_matrix_product



(b) dot_product

Figure 8: **Second example for the recognition of** `matrix_matrix_product` **by the PAP recognizer** This example shows a code excerpt representing a tiled implementation of the matrix-matrix product, together with the result of the recognition process for the whole concept and its subconcepts. This code contains two optimizations, which represent interesting implementation variations: the utilization of a temporary *accumulator* variable (`ACCUM`), and the *tiling* of the loop nest iterations. The first optimization is taken into account within the recognition of the matrix-vector product subconcept, and is represented with a dependence chain (see (b)), the tiling optimization is taken into account within the recognition of the component scans, by means of their strip-mining variant.

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

Table 1: Main differences between the two approaches

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 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 concept instance which annotates the root of the matched subtree. The concept instance contains all the information of what is computed in that subtree (but not any more *how* it is computed). A concept instance I annotating a node v can be used for two purposes: (1) It can be used to determine the concept of the parent node of v. There is no need of I for the recognition process any more after v's parent 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. have generation.

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 concept is one-to-one linked to a node in the abstract syntax tree. This makes concept 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 the application of 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 [5]. The most important transformation in this context is loop distribution because it allows factoring out different computations from the same loop nest that can be matched separately by different concepts (and thus result in separate code being generated).

The PAP Recognizer doesn't need to restructure during recognition because it doesn't need to canonicalize the program with respect to the concept instances. 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 concept instances and assigns shared code portions to the first of them. The intermediate results of the shared code portions used by subsequent concept 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 concepts, because of the nonlevelled and nonlocalized matching mechanism, and because of the possibility to reuse derived subconcepts for other matchings.

Table 1 summarizes the differences discussed in this section.

Figure 9: **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[a] 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.
Application of concept matching techniques[b] has been proposed for the detection of reductions and recurrences within the framework of a formal system for automatic shared memory parallelization.
EAVE[c] is an expert system for interactive vectorization of Fortran programs. It contains a simple pattern matching tool that can uncover order 1 concepts (vector operations, reductions).
Another pattern matching method[d] 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.
A special approach proposed for recurrence detection[e] offers, at considerable computational effort, the recognition of rather general and multidimensional recurrences. Nevertheless, 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[f] 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 idioms and concepts. 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.

---

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

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

[c] see P. Bose: Interactive Program Improvement via EAVE: An Expert Adviser for Vectorization. *Proc. 2nd Int. Conf. on Supercomputing*, pp 119–130, July 1988, ACM Press, New York.

[d] see S.S. Pinter, R.Y. Pinter: Program Optimization and Parallelization Using Idioms. *Proc. of ACM SIGPLAN Principles of Programming Languages*, pp. 79–92, ACM Press, New York, 1991.

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

[f] see G. Sabot, S. Wholey: Cmax: a Fortran Translator for the Connection Machine System. *Proc. 7th Int. Conference on Supercomputing*, pp 147–156, ACM Press, New York, 1993.

Figure 9: **Related work**
(continued)

Similar to the systems compared in this paper is the first phase of a program comprehension system[g] for Fortran programs recently developed at Convex Computer Corp. It works top–down on the PDG and partly uses Snyder's algorithm[a]. Also researchers at Washington State University[h] propose to use program recognition for automatic parallelization.

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[i] represents code and concepts (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[j] 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 rules 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 concept–directed engine derives new higher–level concepts from the existing ones, utilizing plans as inference rules.

[g] see R. Metzger: Automated Recognition of Parallel Algorithms in Scientific Applications. *IJCAI-95 Workshop Program Working Notes: "The Next Generation of Plan Recognition Systems"*, sponsored jointly by IJCAII/AAAI/CSCSI, Aug. 1995.

[h] see S. Bhansali, J.R. Hagemeister, C.S. Raghavendra, H. Sivaraman: Parallelizing sequential programs by algorithm-level transformations. In V. Rajlich and A. Cimitile (editors), *Proceedings of the 3rd IEEE Workshop on Program Comprehension*, Washington, Nov. 1994. pp. 100–107. Los Alamitos: IEEE Computer Society Press.

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

[j] see M. Harandi, J. Ning: Knowledge-Based Program Analysis. *IEEE Software*, pages 74–81, January 1990.

6 Conclusion

We have presented two systems for automatic program comprehension, the PARAMAT concept 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 concept 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 one or the other system for a certain application program, depending on its size and complexity.

We are currently studying the possibility and effectiveness of combining both approaches. One possibility could be the following. If recognition time is very critical, we recommend using PARAMAT's concept 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 the PAP recognizer would take too much time and one is willing to trade recognition power for time, we propose that the PARAMAT recognizer could be run as a preprocessor; the concept instances produced then become facts for the PAP recognizer. Restructuring done by the PARAMAT recognizer is finished before the PDG for the PAP recognizer is constructed. The already recognized program parts are not submitted to the PAP recognizer any more, resulting in a downsized PDG. This corresponds to explicit a priori pruning of the PAP recognizer's 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 concepts. It is up to the user to decide about that when recognition time is critical.

References

[1] B. Di Martino, G. Iannello. Towards Automatic Parallelization through Program Comprehension. In V. Rajlich and A. Cimitile, editors, *Proceedings of the* 3^{rd} *IEEE Workshop on Program Comprehension*, Washington, Nov. 1994. pp. 108–115. Los Alamitos: IEEE Computer Society Press.

[2] B. Di Martino, G. Iannello, H. Zima. An Automated Algorithmic Recognition Technique to Support Parallel Software Development. Proc. of *Int. Workshop on Parallel and Distributed Software Engineering*, Boston (USA), 17-18 May 1997, IEEE-CS press.

[3] B. Di Martino, H. Zima. Support of Automatic Parallelization With Concept Comprehension. *Journal of Systems Architecture*, Elsevier, 1997 (to appear).

[4] 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, Basel, Switzerland, Apr. 1994.

[5] C.W. Keßler. Pattern-Driven Automatic Parallelization. *Scientific Programming* **5**, pp. 251–274, 1996.

[6] C. W. Keßler, H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal on Parallel Programming* **25**(1) pp 17–50, Feb. 1997.

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

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

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