Pattern-Driven Automatic Parallelization

Christoph W. Keßler
Fachbereich IV - Informatik
Universität Trier
D-54286 Trier, Germany
e-mail: kessler@psi.uni-trier.de

Abstract

This paper describes a knowledge–based system for automatic parallelization of a wide class of sequential numeric codes operating on vectors and dense matrices, and for execution on distributed memory message-passing multiprocessors.

Its main feature is a fast and powerful pattern recognition tool that locally identifies frequently–occurring computations and programming concepts in the source code. This tool works also for dusty deck codes that have been 'encrypted' by former machine-specific code transformations.

Successful pattern recognition guides sophisticated code transformations including local algorithm replacement such that the parallelized code need *not* emerge from the sequential program structure by just parallelizing the loops. It allows access to an expert's knowledge on useful parallel algorithms, available machine–specific library routines, and powerful program transformations,

The partially restored program semantics also supports local array alignment, distribution and redistribution, and allows for faster and more exact prediction of the performance of the parallelized target code than is usually possible.

Key words: automatic parallelization, automatic program transformation, algorithm replacement, pattern recognition in numerical codes, automatic data distribution, performance prediction

1 Introduction

Parallel computers with distributed memory are known to be difficult to program. Even more problematic is automatic parallelization for such machines. The most challenging problems that a parallelizing compiler is faced with are the following:

- Parallel code must contain explicit message passing statements. But explicit programming of message passing is complex, tedious and error-prone.
- The efficiency of the target program depends heavily on choosing a suitable distribution (and sometimes, even redistribution) of the arrays occurring in the source program. But for larger applications, this is a difficult global optimization problem.
- A SPMD program generated semi–automatically from a sequential source program by adapting it to given array distributions must in general be transformed to be efficient — just applying the owner-computes-rule will usually not suffice. But there is no guidance on which optimizing transformations to choose, and in which order to apply them. Moreover, there is no possibility to exploit explicitly *parallel algorithms* that have been developed over the last decades for various problems on various target architectures.
- Run time prediction for nontrivial codes on real machines is a very complex issue, due to network contention, message protocols, buffering, undocumented hardware features, and other problems. But reliable run time prediction is essential to estimate the quality of array distribution schemes or of program transformations.

Message passing statements can be generated automatically today by *semiautomatic* parallelization [CK88, ZBG88]. The user has to provide array distributions and optimizing transformations manually, either in the form of interactive commands, as in SUPERB [ZBG88], or in the form of language constructs or compiler directives in an explicitly parallel programming language such as Fortran-D [HKT91], Vienna Fortran [CMZ92], High Performance Fortran [HPF93] and others. Nevertheless, there remains the hard problems involved in automatic data distribution and redistribution, in automatic guidance on optimizing transformations, and in suitably accurate performance prediction.

The problems involved in generating good parallel code for distributed memory multiprocessors (or other complex supercomputer architectures) arise from the fact that there is often not sufficient knowledge available of the source program and on the target machine characteristics. Thus, an automatic parallelizer for such target architectures must be able to acquire and access as much of this knowledge as possible. This does not work for all programs.

Many numerical programs are, however, particularly suitable for this purpose: As a result of considering numerical algorithms in books and courses, and studying a large number of typical application codes that are reasonable candidates to be ported to distributed memory systems, we have observed [Keß94a] that there is only a rather limited number of typical operations, called *patterns*, that often occur in these programs, in particular in the time–consuming inner loops.

These patterns are mostly data parallel operations like elementwise operations on vectors and matrices, various kinds of reductions and linear recurrences, difference stars, grid relaxation sweeps, convolutions and others. A pattern is considered to be a primitive with respect to mathematical properties, data structures of operands, memory access structure, array alignment preferences and run time behaviour. We have collected around 150 patterns in a *basic pattern library*, and also recorded typical implementation prototypes (syntactic variations) of these patterns that are used in sequential source codes [Keß94a].

Based on this observation, we construct an automatic parallelization system called PARAMAT ("PARallelize Automatically by pattern MATching"), with the following key ideas:

- The first step of parallelization must contain a pattern recognition tool that works fast and reliably. Code pieces in the source program that are recognized as an occurrence of one of our patterns are replaced by an *instance* of that pattern, looking similar to a call to an externally defined function. The input language is structured C without pointers.
- Once the system knows what the source program locally does, it can infer additional knowledge using mathematical properties and efficient implementations of the patterns on the target machine, and access offline–generated information on favorable data distributions and run time behaviour of the pattern implementations on the target machine. The parallelization system can then easily use this knowledge to guide a sophisticated parallelization process with high–level program transformations including local algorithm replacement.

The remainder of this paper is organized as follows: Section 2 describes pattern recognition in numerical codes and summarizes our list of patterns. Section 3 presents the main ideas and definitions of our pattern recognition method and gives several examples. Section 4 summarizes the PARAMAT pattern recognition tool, gives results, and discusses some extensions. Section 5 shows how the information supplied by pattern recognition is used to guide automatic parallelization. Section 6 lists some related approaches to pattern recognition and pattern–driven automatic parallelization.

2 Patterns in scientific programs

In order to promote the pattern recognition approach, we have examined many sequential numerical algorithms that are typical and well–suited candidates to be run on distributed memory multiprocessors, e.g. some of the "Numerical Recipes" [PTVF92], algorithms considered in numerical textbooks like [BBC+93] or in a numerical maths course: Basic linear algebra subroutines (see also [LHKK79, DDHH88]), direct

order	patterns	number
0	scalar arithmetics, init, copy, max, min, swap, read, write, etc.	20
	MULTIADD$^{(0)}$, MULTIMUL$^{(0)}$, grid stencil 1D (HSTAR$^{(0)}$) and 2D (STAR$^{(0)}$)	4
1	loop accumulating scalar values (FSUM$^{(1)}$)	1
	elementwise vector operations (VADD$^{(1)}$, VMUL$^{(1)}$, ...), scalar plus vector (VINC$^{(1)}$), scalar times vector (SV$^{(1)}$), full vector triad (VADDSV$^{(1)}$), accumulating vector triad (VAADDSV$^{(1)}$,...), vector init. (VINIT$^{(1)}$, VASSIGN$^{(1)}$, ...), vector copy (VCOPY$^{(1)}$), vector swap (VSWAP$^{(1)}$), vector read/write, etc.	32
	1D reductions: total sum of vector elements (VSUM$^{(1)}$), total product (VPROD$^{(1)}$), inner product (SSP$^{(1)}$, VQSUM$^{(1)}$), etc.	7
	1D reductions: vector maximization/minimizations (value: VMAXVAL$^{(1)}$, VMINVAL$^{(1)}$), location: (VMAXLOC$^{(1)}$, VMINLOC$^{(1)}$), both (VMAXVL$^{(1)}$, VMINVL$^{(1)}$))	6
	1D relaxation steps: Jacobi (VJACOBI$^{(1)}$), Gauss–Seidel (VGAUSSSEIDEL$^{(1)}$)	2
	first order linear recurrences (FOLR$^{(1)}$, PREVSUM$^{(1)}$, SUFVSUM$^{(1)}$)	3
	intermediate form of 1D convolution	1
	global vector update (VLUD$^{(1)}$)	1
	vector shift (VSHIFT$^{(1)}$)	1
2	elementwise matrix operations (MADD$^{(2)}$, MMUL$^{(2)}$, ...), scalar plus matrix (MINC$^{(2)}$), scalar times matrix (SM$^{(2)}$), matrix triad (MAADDSM$^{(2)}$), matrix init. (MINIT$^{(2)}$, MASSIGN$^{(2)}$), matrix copy (MCOPY$^{(2)}$), matrix read/-write, etc.	17
	matrix–vector multiplication (MV$^{(2)}$) and related patterns	3
	forward and backward substitution (FSUBST$^{(2)}$, BSUBST$^{(2)}$)	2
	2D-reductions: total sum of matrix elements (MSUM$^{(2)}$), total product (MPROD$^{(2)}$), concurrent row/col–vector sum (VVSUM$^{(2)}$) or product (VVPROD$^{(2)}$)	4
	2D-reductions: matrix maximizations/minimizations (total or row/col-wise) value (MMAXVAL$^{(2)}$, MMINVAL$^{(2)}$), location (MMAXLOC$^{(2)}$, MMINLOC$^{(2)}$), both value and location (MMAXVL$^{(2)}$, MMINVL$^{(2)}$)	12
	2D relaxation steps: Jacobi (MJACOBI$^{(2)}$, ...), Gauss–Seidel (MGAUSSSEIDEL$^{(2)}$, ...)	4
	global matrix update (MLUD$^{(2)}$, ... intermediate LU decomposition)	3
	1D convolution (VCONV$^{(2)}$); intermediate forms of 2D convolution	3
	matrix shift (MSHIFT$^{(2)}$), row/col-vector-shift (VVSHIFT$^{(2)}$)	2
3	matrix multiplication (MM$^{(3)}$), LU decomposition (LUD$^{(3)}$)	6
	intermediate forms of 2D convolution	2
	2D relaxation loops: Jacobi (JACOBI$^{(3)}$), Gauss–Seidel (GAUSSSEIDEL$^{(3)}$)	2
4	2D convolution (MCONV$^{(4)}$)	1

Table 1: A brief summary of the patterns included into the current version of the Basic PARAMAT Pattern Library. All BLAS routines operating on dense real matrices are included. A pattern's order number (left hand column) denotes the depth of a loop nest that is usually encountered in a straightforward sequential implementation of that pattern. The so–called unstable patterns, e.g. general vector operation (GVOP$^{(1)}$) or multiple vector triad (VMULTIADD$^{(1)}$), are not listed because they are decomposed into their basic patterns before being submitted to the code generation stage, thus being invisible to code generation.

---

solvers for linear equation systems (such as Gaussian Elimination, LU, QR or Cholesky decomposition), Simplex, iterative linear equation solvers (such as Jacobi, Gauß–Seidel, JOR, SOR and Conjugate–Gradient solver), fixpoint iterations (e.g. square–rooting a matrix), grid relaxations (used e.g. for numerical solution of partial differential equations), interpolation problems, numerical integration and differentiation, and multigrid algorithms. These algorithms are the basic building blocks of many numerical applications.

Considering these numerical algorithms in numerics books and courses, and studying a large number of typical application codes as e.g. the Purdue Set benchmark ([MFL+92], see Table 2), the Livermore Loops ([McM86], see Table 3), and others (see [Keß94a]), that are reasonable candidates to be ported to distributed memory systems, we have observed that there is only a rather limited number of typical, mostly data parallel operations, called *patterns*, that often occur in these programs, in particular in the time–consuming inner loops. A pattern is considered to be a primitive with respect to mathematical properties, data structures of operands, memory access structure, array alignment preferences and run time behaviour. We have collected around 150 patterns in a *basic pattern library* (see [Keß94a], chapter 5 for the complete specification; Table 1 gives an overview). We have also recorded typical implementation prototypes (syntactic variations) of these patterns that are used to occur in the sequential source codes considered; they are specified in appendix B of [Keß94a].

Our observations are backed up by other empirical investigations on large `FORTRAN` codes [SLY90] and by the typical sets of numerical routines contained in numerical linear algebra packages, which are either supplied by hardware vendors, or offered by numerical software companies, or distributed as public domain software.

So far, we have focussed on algorithms operating on rectangular dense real matrices since these are the most reasonable candidates to be ported to distributed memory parallel supercomputers; nevertheless, our approach may easily be extended to other matrix types (e.g., banded, block–banded; complex). We are currently investigating operations on sparse matrices [Keß94a].

No.	Name	recognized patterns	recognized loops
1	Trapezoidal rule	`FSUM`	1 from 1
2	reduction function 1	`MINITSP, VVPROD, VSUM`	3 from 3
3	reduction function 2	`MINIT, VVPROD, VSUM`	3 from 3
4	reduction function 3	`VINIT, VINV, VSUM`	3 from 3
5	simple search	`MINITSP, MSUM, -`	2 from 3
6	tridiag. set of lin. eqns.	`VINIT` (8), `VMUL` (4), `GVOP` (8), `VCOPY` (5), `VSUM`	26 from 26
7	Lagrange interpolation	`VINITSP`(2), `VINC` (2), `VINV`, `VPROD` (2), -	7 from 8
8	divided differences	`VINITSP, VSIN, -, MSUM`	3 from 4
9	finite differences	`MINITSP, MINIT, MJACOBI, -, MCOPY, MSUM`	5 from 6
11	Fourier's moments	`VINITSP, GVOP, VSUM`	3 from 3
12	array construction	`VINITSP` (2), `MINITSP, MCOPY, VCOPY` (2)	6 from 6
13	floating point arithmetic	`VINITSP, GVOP` (3), `VMULTIADD`, `VCONDASS(VADD), VQSUM`	7 from 7
14	Simpson's and Gauss' integration	`FSUM` (5)	5 from 5
15	Chebyshev interpolation	`VINITSP` (2), `GVOP` (3), `VCOPY, -`	6 from 7

Table 2: Analysis of the Purdue Set (sequential versions of 14 kernels from the HPF benchmark suite [MFL+92]): currently recognizable patterns. The right–hand column indicates how many loops (after applying loop distribution) can be covered by patterns from the Library. `GVOP` denotes a general vector operation that is later decomposed into atomic elementwise vector operations using temporary arrays.

3 Principles of pattern recognition

3.1 Overview

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.

The algorithm traverses the abstract syntax tree from left to right in postorder. For a leaf node (a variable or a constant), determining its pattern is trivial (VAR or CONST, respectively). At each inner node v of the syntax tree, it tests, based on 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. This is technically arranged by calling a short routine, a realization of a so–called *template*. This routine fails if it cannot prove that the function computed by T_v equals the operation represented by m. Otherwise, it returns an instance I of pattern m, maps the program objects to the corresponding slots of I, and annotates v with I. If there are several templates admissible, these are tested concurrently (the result is deterministic). Failing templates abort as soon as possible.

The already matched patterns of v's children dramatically prune the search space of patterns that may match v. Often, there is already one characteristic pattern (*trigger pattern*) of a child of v together with v's operator to select a single possible template. We give the formal definitions of these concepts in subsection 3.3.

This (classical) pattern matching along 'vertical' edges of the abstract syntax tree corresponds to a special deterministic bottom-up tree automaton [FSW92]. This procedure can be 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. Several instructions may belong to the same computation only if their operands are involved in at least one of several types of dataflow relations. We denote important dataflow relations by dataflow edges (*cross edges*). Computation of these edges (i.e., computing exact array data flow) is generally hard, but in our case, we can profit from the simple array access structures that are characteristic for dense matrix computations and that are present in all our patterns. We will consider this problem in subsection 3.5.

3.2 Preparing Code Transformations

Before starting pattern recognition, we apply several important normalizing transformations to make the program as explicit as possible, by

- inlining all procedures (recursive procedures are very untypical for the application area considered); this makes all program analysis intra–procedural,
- forward propagation of constant expressions,
- making control flow well–structured by eliminating `goto`s,
- recognition and replacement of induction variables (i.e., integer variables indexing arrays that are not a loop variable of a surrounding `for` loop) by a term depending only on loop variables, and
- eliminating dead code.

These transformations are applied in this order just once (regarding ordering of transformations, see [WS90]).

3.3 Patterns, Templates, and the Pattern Hierarchy Graph

Each nontrivial pattern m is a pair (f_m, ℓ_m) consisting of a specification f_m of a (mathematical) operation, and a list ℓ_m of specifications of the types and the data structures of the parameters occurring in f_m.

For instance, the $\mathsf{MV}^{(2)}$ 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 m, we usually know several implementation prototypes (for sequential C code). Because of the wide variety of semantics preserving code transformations, the number of such prototypes can be large for more complex patterns (such as matrix-matrix-multiplication), expanding 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-Multiplication ($\mathsf{MV}^{(2)}$) can be written as a single loop based on a dot product computation

```
for (i=1; i<=n; i++)
   SSP(j=[1:m], 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(i=[1:n], 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 instances.

Recognizing leaf nodes in the syntax tree as variables or constants is trivial. Now consider a subtree T_w rooted at a node w with several children $v_1, ..., v_k$. The operator *op* of w is either a `for` loop header, an `if` header, an assignment, or a unary or binary expression operator. The children of w respectively correspond to the loop body, the `then` or `else` branch, the left hand side variable or the right hand side expression of the assignment, or the operand expressions.

Definition Let h be the function computed by T_w, as defined by the semantics of the programming language used. Let the children $v_1, v_2, ..., v_k$ of node w already being annotated by pattern instances $I_1, I_2, ..., I_k$ of (potentially, trivial) patterns $m_1, m_2, ..., m_k$ from the library. Let g denote a function. Let $i \in \{1, ..., k\}$.

We call the $k+2$–tuple $S = (g, m_1, ..., m_k, i)$ a *template* of m, if $g(f_{m_1}, ..., f_{m_k}) = f_m = h$.

We call m_i a *trigger pattern*; i is, depending on *op*, determined according to the table below.

Moreover, we call $m_1, ..., m_k$ (potential) *subpatterns* of m. $\square$

For each pattern, we realize only the most important templates (typically, we have 1 to 3 realized templates per pattern), see [Keß94a]. The trigger pattern is chosen as follows:

operator *op* of node w	child of w carrying the trigger pattern
`for` loop header	loop body (first statement)
`if` header	then part (first statement)
assignment	root of right hand side expression
expression operator	left or right subexpression

Definition A *pattern hierarchy graph* (PHG) for a set $\mathcal{M}$ of patterns m is a directed graph $G = (V, E)$. The set V of nodes contains all patterns $m \in \mathcal{M}$. For each realized template $S = (g, m_1, ..., m_i, ..., m_k, i)$ for a pattern m with trigger pattern m_i there is an edge (m_i, m) in E. $\square$

Since $m_i = m$ is possible (i.e. a pattern may occur as a subpattern in one of its own templates), there may exist trivial cycles from a pattern to itself. Apart from these trivial cycles, the PHG is *acyclic*.

We associate an order number *order*(m) with each pattern m that denotes the loop nesting depth in a straightforward sequential implementation of m (i.e., without blocked loops). For example, for matrix-vector multiplication, we have *order*$(\mathsf{MV}^{(2)}) = 2$, and for matrix-matrix multiplication, we have *order*$(\mathsf{MM}^{(3)}) = 3$. A PHG edge (m_i, m) implies *order*$(m_i) \leq$ *order*(m).

A PHG is called *complete* for a pattern m, if its node set contains m and all subpatterns $m_1, ..., m_k$ of m occurring in any realized template of m, and if it is complete for all m_j, $1 \leq j \leq k$.

It follows that the PHG complete for a subpattern m_j of m is a subgraph of the PHG complete for m.

If m_i is a trigger pattern in some template of m, we call m a *superpattern* of m_i. We denote by $SP(m_i)$ the set of all superpatterns of m_i. Usually, a pattern has only a small number of superpatterns (see [Keß94a]). Let w be as above, then the set of possible candidate patterns that may match w is

$$\bigcap_{\substack{1 \leq j \leq k \\ SP(m_j) \neq \emptyset}} \{m : (m_j, m) \text{ edge in PHG}\} \tag{1}$$

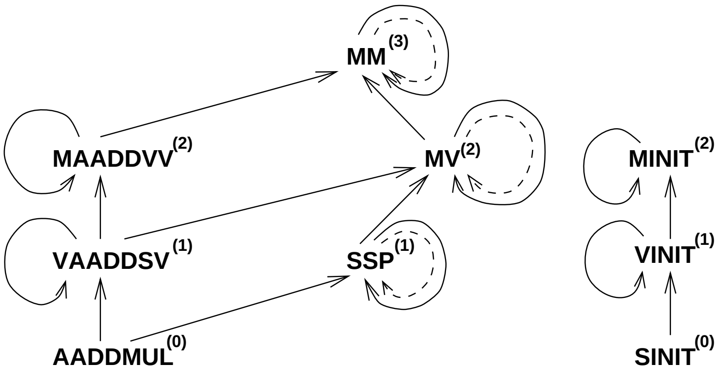


Figure 1: The pattern hierarchy graph of Matrix–Matrix–Multiplication. Solid edges mean realized templates for 'vertical' pattern recognition; dashed edges for 'horizontal' pattern recognition along cross edges. Solid cycles mean templates for unblocking or elimination of semantically invariant conditionals;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

---

and the set of templates of these patterns that are to be tried out at w is determined analogously.

Thus pattern recognition becomes a path finding problem in the PHG. Different paths towards a pattern m correspond to different implementations of the functionality of m. This means that a linear–sized PHG (and thus, pattern recognizer) represents exponentially many implementation variations of the same pattern.

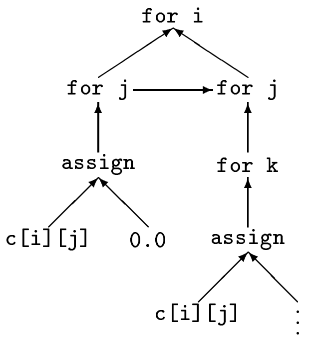
The PHG has a second important advantage: it serves as a hash table that can be inspected by the pattern recognition algorithm, because it yields all the possible superpatterns that could be matched from a given trigger pattern. Often, the trigger pattern together with the operator of the node to be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

3.4 Examples

Matrix–Matrix–Multiplication We demonstrate the pattern recognition algorithm using a simple example. Matrix–Matrix–Multiplication is well suited since its functionality and subpatterns is widely known. Its PHG is given in Figure 1.

Suppose the programmer has coded Matrix–Matrix–Multiplication as follows:

```
  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 pattern recognition 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( j=[1:m], c[i][1:m], 0.0)`. The access to array `c` has become a vector, since one dimension has been bound by the loop.

Then, the algorithm considers the assignment `S2` and annotates it 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 (k=[1:r], 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 $\mathsf{SSP}^{(1)}$ instance to keep data access information consistent. In the next step, the `do j` loop around the $\mathsf{SSP}^{(1)}$ instance is recognized as an instance of matrix–vector–multiplication. Also in this case, 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):

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

At this stage, we can continue pattern recognition only if we take care of data flow. Exact array dataflow analysis, although generally a very hard problem [Fea91, MAL93], is dramatically simplified by the exact data access information supplied with the pattern instances. In this example, we find that the vector `c[i][1:m]` is written in the $\mathsf{VINIT}^{(1)}$ instance, and read and overwritten by the $\mathsf{MV}^{(2)}$ instance, symbolized by a so–called *cross edge* of type FLOW. Thus, we have exact information 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, we can merge these two instances into a single $\mathsf{MV}^{(2)}$ instance `MV (k=[1:r], j=[1:m], 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 $\mathsf{VINIT}^{(1)}$ instance. This instance, in turn, can be matched with the `i` loop into `MM(k=[1:r],i=[1:n],j=1:m], 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 $\mathsf{SINIT}^{(0)}$...$\mathsf{VINIT}^{(1)}$ and $\mathsf{AADDMUL}^{(0)}$... $\mathsf{SSP}^{(1)}$...$\mathsf{MV}^{(2)}$...$\mathsf{MV}^{(2)}$...$\mathsf{MM}^{(3)}$. Common program transformations, like loop interchange or loop distribution, would result in a different path being taken towards $\mathsf{MM}^{(3)}$, but would not prohibit pattern recognition.

Elimination of semantically redundant conditionals The following fragment is taken from the `MATMUL` routine of the DYFESM program from the Perfect Club Benchmark Suite [Ber92]:

```
       DO 300 J = 1, M
         IF (B(J,K) .NE. 0.) THEN
         DO 200 I = 1, L
           C(I,K) = C(I,K) + A(I,J)*B(J,K)
200      CONTINUE
300    CONTINUE
```

The programmer has added the condition `IF (B(J,K).NE.0.0)` to avoid unnecessary multiplications and additions by 0.0. Since the program's semantics is not changed by this optimization, we realized a new template for the vector triad $\mathsf{VAADDSV}^{(1)}$ (and for several similar patterns) that follows a self cycle in the PHG to remove the condition, just by copying the $\mathsf{VAADDSV}^{(1)}$ instance at the `I` loop header to its parent node, the `IF` header. Pattern recognition then proceeds as above.

Unblocking loops Blocked loops are very common in dusty deck programs that have been optimized for other target architectures with caches or vector registers. In the following example, the `i` loop has been blocked by a factor of `k`:

```
for (i=1; i<=n; i+=k)
  for (j=i; j<=min(n,i+k-1); j++)
    dy[j] = dy[j] + da*dx[j];
```

The inner loop is recognized as a $\mathsf{VAADDSV}^{(1)}$ instance:

```
for (i=1; i<=n; i+=k)
  VAADDSV(j=[i:min(n,i+k-1)], dy[i:min(n,i+k-1)], da, dx[i:min(n,i+k-1)], dy[i:min(n,i+k-1)]);
```

Another template (corresponding to another PHG self cycle) discovers that the `i` loop is blocked, and annotates it by

```
VAADDSV(i=[1:n], dy[1:n], da, dx[1:n], dy[1:n]).
```

Similar unblocking templates exist for many other elementwise vector and matrix operations and for many reductions. The normalizing transformation 'loop unblocking' has thus been integrated into the pattern recognizer as a realization that is shared by all these templates. This integration is possible since the syntax tree structure is not modified. However, this does not hold for *loop distribution* (a loop transformation important for pattern recognition) which has to be called separately before each recognition step.

Difference stars $\mathsf{MULTIMUL}^{(0)}$ matches a multi–operand product of scalars; $\mathsf{MULTIADD}^{(0)}$ matches a multi–operand sum of scalars or products of scalars. The trigger patterns for $\mathsf{MULTIADD}^{(0)}$ are ADD, $\mathsf{ADDMUL}^{(0)}$, $\mathsf{MULMUL}^{(0)}$, $\mathsf{MULTIMUL}^{(0)}$ and $\mathsf{MULTIADD}^{(0)}$; these for $\mathsf{MULTIMUL}^{(0)}$ are MUL and $\mathsf{MULTIMUL}^{(0)}$. Distributivity is not applied. Double negations (`-(-a))`) or inversions (`1/(1/a)`) are eliminated. Subtractions are represented as sums, divisions as products. Negations and inversions are represented as flag bits in their operand nodes; this makes expression trees more compact and easier to recognize.

Difference stars (stencils), in one ($\mathsf{HSTAR}^{(0)}$) and two ($\mathsf{STAR}^{(0)}$) dimensions, are the most important building blocks of grid relaxation sweeps. They are always based on an ADD, AADD or a $\mathsf{MULTIADD}^{(0)}$. The following Gauss–Seidel relaxation (Livermore Loop 23)

```
for (j=2; j<=6; j++)
  for (i=2; i<=N; i++)
    ZA[i][j] = ZA[i][j] + 0.175 * ( ZA[i][j+1]*ZR[i][j] + ZA[i][j-1]*ZB[i][j]
                                  + ZA[i+1][j]*ZU[i][j] + ZA[i-1][j]*ZZ[i][j]
                                  - ZA[i][j] );
```

contains a five-point-stencil. The realization of the $\mathsf{HSTAR}^{(0)}$/$\mathsf{STAR}^{(0)}$ templates refines the just recognized $\mathsf{MULTIADD}^{(0)}$ instance to a $\mathsf{STAR}^{(0)}$ instance and calls itself as long as further, optional $\mathsf{STAR}^{(0)}$ parameters can be filled in:

```
for (j=2; j<=6; j++)
  for (i=2; i<=N; i++)
    STAR ( ZA[i][j], _ , _ , 0.175000, _ , _ , _ , ZA[i][j], _ , _ ,
           _,            ZB[i][j],     _,
           ZZ[i][j],     4.714286,     ZU[i][j],
           _,            ZR[i][j],     _,
           i,1,1, j,2,1);
```

Now, further recognition of $\mathsf{MGAUSSSEIDEL}^{(2)}$ is straightforward.

3.5 Exploiting the cross edges

Cross edges in the syntax tree represent particular, loop-independent data flow relations among the operands of pattern instances within the same block. Pattern instances interconnected by a cross edge may, even if textually separated, belong to the same thread of computation, and thus, to the same superpattern. Therefore, cross edges are well–suited to guide 'horizontal' pattern recognition.

In [Keß94b], we have devised a compact array access descriptor that supports fast realizations of the important query operations equality, inclusion, disjointness and (direct) neighbourhood of array access shapes. A descriptor is computed for each operand of a pattern instance just after generating it. Thus, only one loop level has to be considered at a time. Furthermore, each operand has one of four possible access modes: I (ignore), R (read), W (write), RW (read and overwrite). For non-recognized code fragments, worst case assumptions have to be made. From this information, we easily compute five different types of cross edges that are important for pattern recognition. A *cross edge* connects an instance I_1 to an instance I_2 located textually behind I_1 within the same block, and has type

1. FLOW if I_1 writes an object that is read by I_2, and this data flow is not killed by another instance I_3 located between I_1 and I_2 that writes to this object; this corresponds to a loop-independent data flow dependence from I_1 to I_2.

2. ANTI if I_1 reads an object that is written by I_2, and this data 'flow' is not killed by another instance I_3 located between I_1 and I_2 that writes to this object; this corresponds to a loop-independent data anti dependence from I_1 to I_2.

3. INPUT if both I_1 and I_2 read the same object that is not written to by another instance I_3 located between I_1 and I_2.

4. NEIGHBOUT if I_1 and I_2 write neighboured sections of the same object that are not read or written by another instance I_3 located between I_1 and I_2.

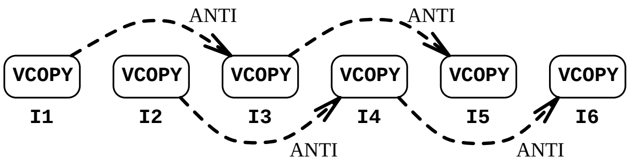
5. NEIGHBIN if I_1 and I_2 read neighboured sections of the same object that are not written to by another instance I_3 located between I_1 and I_2.

In general, the cross edges of a block form a directed acyclic graph.

Only pattern instances connected by cross edges are considered for a potential merge in pattern recognition. Selection of suitable templates is guided by the type of the cross edge and by the (trigger) pattern name of the last pattern instance (I_2). If several templates should be admissible, then they can be tried out concurrently; at most one of them may really match, thus determinism is preserved.

Cross edges of type ANTI are used at recognition of VSWAP$^{(1)}$ from three single VCOPY$^{(1)}$ (vector copy) instances:

```
I1: VCOPY(i=[1:n],t1[:], a[:]);
I2: VCOPY(i=[1:n],t2[:], b[:]);
I3: VCOPY(i=[1:n], a[:], c[:]);
I4: VCOPY(i=[1:n], b[:], d[:]);
I5: VCOPY(i=[1:n], c[:],t1[:]);
I6: VCOPY(i=[1:n], d[:],t2[:]);
```



Instances belonging to the same VSWAP$^{(1)}$ computation are chained by ANTI cross edges.

The interleaving of the instances does not prohibit the recognition process since it is guided by the cross edges. We obtain

```
I1': VSWAP(i=[1:n], a[:], c[:], t1[:]);
I2': VSWAP(i=[1:n], b[:], d[:], t2[:]);
```

The following special cases of pattern matching along cross edges are particularly important for us:

- **Loop rerolling:**

  Loop unrolling is a common program optimization. It occurs (1) as replication on the expression level (within the same expression) and (2) as replication on the statement level (different statements in the same block). When rerolling loops, in general, several instances are merged at once. These instances form a connected component of cross edges of type NEIGHBIN or NEIGHBOUT (see [Keß94b]).

- **Renaming/removing of temporary variables:** Often, reduction implementations use temporaries for the accumulating variables, e.g., to enforce register usage, or to avoid complicated addressing:

  ```
  for (i=1; i<=n; i++) {
    SSP( j=[1:m], temp, a[i][1:m], b[1:m], 0.0 );
    SCOPY( x[i], temp );
  }
  ```

  Immediately after recognition of $\mathsf{SCOPY}^{(0)}$ and computation of cross edges, the FLOW cross edge from the $\mathsf{SSP}^{(1)}$ to the $\mathsf{SCOPY}^{(0)}$ instance selects a $\mathsf{SSP}^{(1)}$ template that replaces the temporary `temp` by `x[i]` and removes the (now useless) $\mathsf{SCOPY}^{(0)}$ instance.

  Due to the one-pass nature of the pattern recognition algorithm, we do not know at this point whether the last value of `temp` (i.e., the `n`th component of vector `x`) may be used later on. Thus, to maintain consistency, we insert a correcting $\mathsf{SCOPY}^{(0)}$ instance. After loop distribution and one further pattern recognition step, we have

  ```
  MV( i=[1:n],j=[1:m], x[1:n], a[1:n][1:m], b[1:m], 0.0 );
  SCOPY( temp, x[n] );
  ```

  The $\mathsf{SCOPY}^{(0)}$ instance may later be removed as useless code if `temp` is not used any more.

3.6 The pattern recognition algorithm

The function *stmtdescend*() traverses the syntax tree in postorder; *exprdescend*() does the same for expression trees (where, however, no cross edges can occur).

---

function *stmtdescend*(*node*)
if *node* is already visited **then return fi**
if *node* is not an assignment statement **then forall** children s of *node* **do** *stmtdescend*(s) **od fi**
forall expressions e occurring in *node* **do** *exprdescend*(e) **od**
/⋆ Now all subtrees of *node* are visited and (perhaps) recognized ⋆/
if *node* is an `IF` header **then** *try_IF_distribution*(*node*) **fi**
if *node* is a `for` loop header **then** *try_loop_distribution*(*node*) **fi**
forall admissible 'vertical' superpatterns m for *node* in the PHG (cf. formula 1)
do test by the 'vertical' template *match*(m,*node*), if there is an instance I of m matching *node* **od**
if not, **return fi** /⋆ FAILED ⋆/
annotate *node* with I; compute access descriptors and cross edges to I
repeat
    forall direct cross predecessors x of *node* (in the same block)
    do /⋆ x has already been visited earlier ⋆/
        test by admissible cross templates if the sequence x; *node* is an incarnation of a superpattern m'
        if yes, merge x and *node*, call the result *node* and annotate *node* with an instance I' of m'; **break**;
    od
until there are no mergeable cross predecessors of *node* left.
end *stmtdescend*()

The routine *try_IF_distribution* tries to distribute a masked block of statements; *try_loop_distribution* tries, for a loop over a block of statements, to perform scalar and vector expansion and thereafter distribute the loop as far as possible (cf. [ZC90]). `IF` distribution and loop distribution modify the structure of the syntax tree: *node* gets several "younger" brothers (copies of *node*) and moves some of the statements from its body to theirs. After *node*, pattern recognition visits the new brother nodes as if these would exist already from the beginning; but revisiting their children (that were children of *node* before and thus are already matched) is not required.

Because of the deterministic nature of the method, each node is visited only once. Since selection of admissible templates is very fast due to PHG inspection, run time cost is dominated by the linear tree traversal time. Data flow is computed by need, i.e. only for the current loop level. Loop distribution uses Tarjan's algorithm for strongly connected components; its pseudocode can be found in [ZC90]. [1]

3.7 Recognition of data structure concepts

Beyond annotating nodes with pattern instances, pattern recognition offers the possibility to keep track of static relations of single program objects. An illustrative example is the identification of statically known grid hierarchies in multigrid programs. Detection of such grid hierarchies is especially important when data is stored in a one-dimensional workspace array. Then, the additional information allows reconstruction of the different two-dimensional grids, supporting array partitioning and load balancing.

3.8 Transformations after pattern recognition

After pattern recognition, we must eliminate useless code that may emanate from conservative cross matching and certain transformations. Useless code computes variables that are not consumed or output before being recomputed.

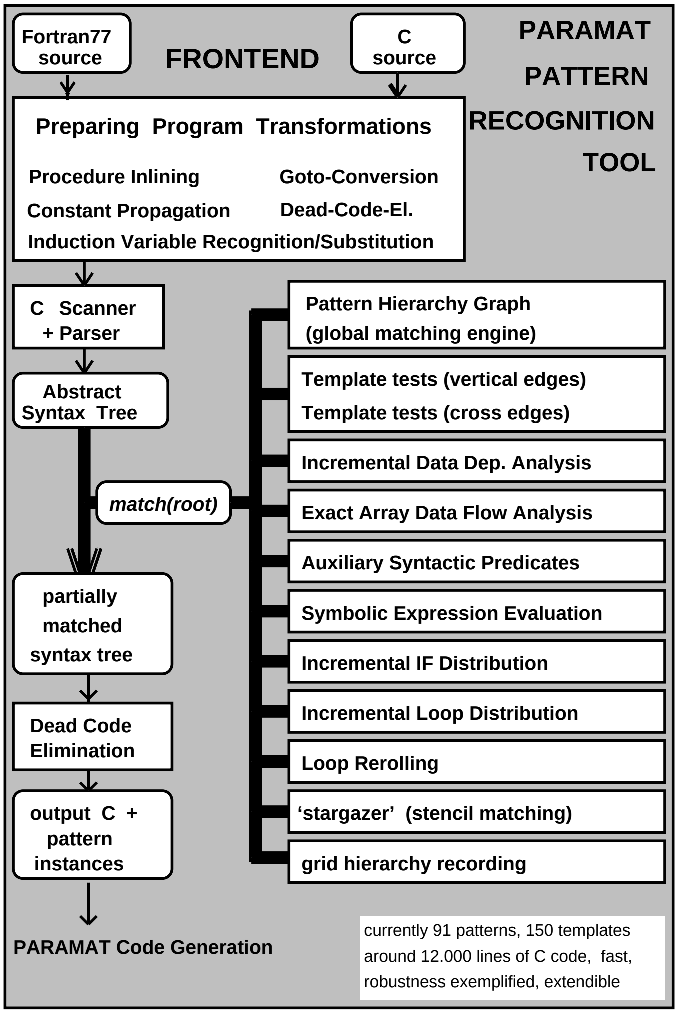
Instances of so–called *unstable patterns* are decomposed into their basic patterns' instances, e.g., the instance `SVSUM(i,c, a, b[1:n], 0.0)` is split into the sequence `VSUM (i, temp, b[1:n], 0.0); MUL(c, a, temp)`. This extraction of a loop-invariant multiplication is a target-machine independent optimization. Furthermore, the number of patterns that are visible for the code generation phase is additionally reduced.

[1] Computation of the data dependency graph for a block of k statements takes, depending on the dependence tests used, in the worst case at least time $O(k^2)$; the data dependency graph itself may require space $O(k^2)$ which is then the input size for Tarjan's linear–time algorithm. This works fast for blocks of moderate size, but, of course, ruins the otherwise linear run time of our algorithm. We tolerate this because blocks tend to be small compared with the size of the entire source program, and because loop distribution is crucial for the robustness of our method.

4 The PARAMAT Pattern Recognition Tool

4.1 Implementation

A prototype of the pattern recognition tool (see figure on the right) has been implemented and tested. The current implementation consists of around 12000 lines of C code and reliably recognizes 91 nontrivial patterns with around 150 nontrivial templates. Each template is implemented as a C routine of around 20 to 50 lines that tests syntactic and semantic conditions and, if successful, generates the pattern instance and fills in the slot entries. Since many useful syntactic and semantic predicates have been predefined, writing code for templates is handy and straightforward. More patterns can easily be added. The high degree of robustness against loop interchange, loop distribution, loop unrolling and statement reordering has been exemplified in practice.



4.2 Results

A *phase* (cf. [BKK93]) is a minimal set of loops around some assignment statements such that all indexing variables occurring in these statements are bound by loop variables.

Ideally, all phases of a program have been recognized completely as incarnations of our patterns.

The pattern recognition tool recognizes nearly all phases in 16 of the 24 Livermore Loops (see Table 3). The recognition times are pretty fast although measured on a low-end Sun SLC, including the time for parsing the source and printing the result. Further encouraging results have been obtained for many other source programs; most of them are listed in the appendix of [Keß94a].

4.3 Discussion

A possible alternative to our syntax-tree-based approach may be pattern recognition on the control flow graph (CFG). We state:

loop	computation	recognized patterns	rec. loops	nodes	time
1	Hydrofragment	GVOP	1 of 1	47	0.2 sec.
3	Inner Product	SSP	1 of 1	35	0.2 sec.
5	tri-diag. elim., below diagonal	FOLR	1 of 1	45	0.1 sec.
7	equation of state fragment	GVOP	1 of 1	88	0.3 sec.
8	A.D.I Integration	VJACOBI (3), GVOP (3)	6 of 6	320	1.3 sec.
9	Numerical Integration	GVOP	1 of 1	91	0.3 sec.
10	Numerical Differentiation	VCOPY (10), VADD (9)	19 of 19	242	1.1 sec.
11	First Sum	PREVSUM	1 of 1	48	0.2 sec.
12	First Difference	VJACOBI	1 of 1	32	0.1 sec.
13	2D particle in a cell	VCOPY (4), VAMOD (4),VAINC (2), VAADD (2)	12 of 17	258	0.9 sec.
14	1D particle in a cell	GVOP (3), VCOPY, VADD (2)	6 of 12	229	0.7 sec.
18	2D explicit hydrodyn. fragment	GMOP (4), MAADDSM (2)	6 of 6	608	2.5 sec.
21	Matrix Product	MM	1 of 1	58	0.1 sec.
22	Planckian Distribution	GVOP (2)	2 of 2	80	0.2 sec.
23	2D implicit hydrodyn. fragment	MGAUSSSEIDEL	1 of 1	105	0.2 sec.
24	1D Minimization	VMINLOC	1 of 1	47	0.1 sec.

Table 3: Livermore Loops [McM86]: 16 of the 24 kernels are (mostly completely) recognized. The fourth column indicates how many loops (counted after applying loop distribution) were matched. The fifth column gives the number of nodes of the abstract syntax tree; the last column the overall times for parsing, recognition and output, measured on a low end SUN SLC, that are quite encouraging.

- The syntax tree representation is supplied by the front end. Since we only admit C statements that produce well–structured control flow, the syntax tree contains all required control dependency information.
- The CFG is much less structured than the abstract syntax tree. By converting the syntax tree in a CFG, we would loose information e.g. about the loop structure (loop variables). Pattern recognition would be harder, less clear, and slower.
- The CFG may be more useful if the source program contains many jumps ('spaghetti code'). For our patterns, however, jumps are rarely required, and can always be replaced by structuring constructs like `IF-THEN-ELSE` or `WHILE`.

Future extensions to the pattern recognizer could address interprocedural matching which would handle recursive functions (that are encountered in many FFT programs), and indirect array references and pointers (that are required for recognition of operations on sparse matrices).

Pattern instances could also be written directly by the programmer in the source text (very similar to Fortran 90's array operations and intrinsic array manipulation functions), thus locally bypassing pattern recognition.

5 Pattern-Driven Parallel Code Generation

The matched intermediate representation is machine independent and opens access to very sophisticated program transformations. Instances of recognized patterns can now be replaced by their best known parallel implementation. These implementations are machine dependent and are parameterized by problem sizes and data distributions of the operand arrays occurring in the instance. They should be written in C with inline assembler for optimal usage of local processor features. Since we want to optimize each pattern implementation only once, off-line at compiler generation time, we assume that the following machine parameters are known at compiler generation time:

- the number of processors,

- sizes of local memory and communication buffers,
- average communication overhead and latency,
- cache size and caching strategy, if existing,
- length of arithmetic pipelines and/or vector registers of the node processor, if they exist.

In principle, there are now two possibilities to generate parallel code for a recognized subtree of the abstract syntax tree: The first alternative is the generation of a standard parallelization according to well–known techniques that we shortly revisit in subsection 5.1 and modify for our purpose in subsection 5.2. The second option, considered in subsection 5.3, addresses the selection of an alternative parallel implementation that computes the same function as the standard parallelization but applies a different parallel algorithm.

5.1 Generation of standard parallel implementations

For given array distributions, a standard parallel implementation is generated according to the following, well–known technique (cf. e.g. [ZBG88]):

(1) *Splitting*: If the target machine has a host that handles all I/O operations, then a host program is generated that performs all I/O operations, starts the node programs on each processor, sends portions of read operands to the node processors that need them, and collects the result values from the node processors that generate them.
(2) *Adaptation*: The node program maintains, in principle, the program structure of the sequential version. For a given partitioning of the arrays, each assignment statement will be masked by a condition depending on the node processors's ID number that ensures that a node processor only executes this statement if it owns[2] the variable on the left hand side of the assignment. Furthermore, interprocessor communication (`EXCH`–statements, cf. [ZBG88]) must be generated to ensure that non-local operands are available when the statement is executed. There is no explicit synchronization needed if blocking `receive` statements are used.
(3) *Optimization*: The masks can often be integrated into the bounds of a surrounding loop, thus avoiding much of the overhead due to the condition evaluation. Interprocessor communication is moved to the topmost loop level (loop distribution) that is still possible without violating data dependencies. Communication is vectorized as far as possible.

The standard parallelization for a *single* loop l with body r consists of a specialization of this scheme: If l indexes the dth dimension of an array occurrence $A[...]$ in r, the dimension-specific mask $\mathit{owned}_d(A[...])$ has to be used instead of $\mathit{owned}(A[...])$, and the dimension-specific communication statement $\mathtt{EXCH}_d(A[...])$ instead of `EXCH` as described above.

In contrast to an explicitly parallel algorithm, the standard parallelization preserves the structure of the sequential program.

5.2 Selection of parallel implementations

For the matched nodes v in the abstract syntax tree, there exist several possibilities to generate code for T_v beyond standard parallelization. The PARAMAT user may a priori control the selection process for each pattern m by setting code generation switches SeqDebug[m], ReplSeq[m], and NoReplace[m]. Based on these switches, at each node v with pattern $m = v.pat$ matched at v, PARAMAT selects among the following alternatives:

(1) a *sequential implementation* $\Delta[m]$ for m (computes m on one node processor or on the host, including the necessary communication), if the debugging bit SeqDebug[m] has been set. The parameters controlling the data distribution and the problem sizes are ignored.

[2] A variable (e.g., a section of an array) is *owned* by a processor if that variable resides in its local memory due to the given data distribution. Scalars are, in general, replicated, i.e. owned by all processors.

(2) a *replicated sequential implementation* $\Xi[m]$ for m (sequential computation on all node processors, corresponding to the given array distributions) if the sequentialization bit $\textsc{ReplSeq}[m]$ has been set. The parameters controlling the data distribution are ignored.

(3) a standard parallel implementation $\Psi[m]$ (see above) for the topmost loop l occurring in T_v, if the bit $\textsc{NoReplace}[m]$ has been set. The implementation chosen for the body r of l depends on the bits for the pattern *r.pat*.

(4) a *parallel algorithm* $\Pi[m]$ for m that is not a standard implementation, if such an algorithm exists. This parallel implementation is also parameterized in data distribution and problem sizes.

The construction of $\Psi[m]$ deserves some clarification. Let L denote the set of loop headers $l \in T_v$ that fulfil *l.pat*$=m$ (i.e., nodes in L are annotated with the same pattern name as v). The loop nesting structure in T_v, as originally programmed, is still available. Several loop headers in L arise from unrolled or blocked loops outside L. Let R be the body of the innermost loop $l_{inn} \in L$. Let $L' \subset L$ the set of loops that block a loop in R. Technically, we make $L' \cup R$ contiguous by interchanging[3] all loops $l' \in L'$ "downwards" with the next inner loop $l \in L - L'$, such that T_v now consists of a contiguous set $L - L'$ of outer loops around a new body R', consisting of the loops of L' around R. If $R' - R \neq \emptyset$, pattern recognition has to be called again for the nodes in $R' - R$ to update the pattern instances for the loop headers in $R' - R$. The same holds for $L - L'$, if some loop had been interchanged. The structure of R remains unchanged. For all loops $l \in L - L'$, a standard implementation is generated.

Let r' denote the root of R'. The code generation method chosen for $T_{r'}$ depends on the code generation switches for the (maybe updated) pattern *r'.pat*.

The effect of $\textsc{NoReplace}[m]$ is thus the same as if the loops in $L - L'$ would not have been recognized as pattern instances (but these in R' would).

Example: Pattern recognition has identified the following code fragment

```
for (i=1; i<=n; i+=x)
  for (j=1; j<=m; j++)
    for (k=i; k<=min(i+x-1,n); k++)
      for (l=1; l<=r; l++)
        a[j][k] = a[j][k] + b[j][l]*c[l][k];
```

as an occurrence of matrix–matrix–multiplication and annotated the `i` loop header $v = l_i$ with the $\mathsf{MM}^{(3)}$ instance

```
MM(j,i,l, a[:][:],b[:][:],c[:][:],a[:][:]).
```

Also the `j` loop header (call it l_j) has been annotated with a $\mathsf{MM}^{(3)}$ instance because the `i` loop only blocks the `k` loop. Thus we have $L = \{l_i, l_j\}$ and $L' = \{l_i\}$. Let us further assume that the PARAMAT user has set $\textsc{NoReplace}[\mathsf{MM}^{(3)}]$. Since l_i blocks another loop (l_k), we interchange it towards the "body" (with l_j) and obtain

```
for (j=1; j<=m; j++)
  for (i=1; i<=n; i+=x)
    for (k=i; k<=min(i+x-1,n); k++)
      for (l=1; l<=r; l++)
        a[j][k] = a[j][k] + b[j][l]*c[l][k];
```

We recognize that, after resubmitting this code to pattern recognition, only the pattern instance of l_i would change (namely, into a $\mathsf{MV}^{(2)}$ instance). That is why we call pattern recognition again only for $R' - R = \{l_i\}$, with the $\mathsf{MV}^{(2)}$ instance at l_k being already given.

Standard parallelization then yields

[3] This loop interchange is generally possible, since for blocking of interchangeable loops similar conditions hold as for loop interchange (the blocking loop does not index any array references) — otherwise, our pattern recognition algorithm would not have recognized l' as a blocking loop. — As an alternative, we also may explicitly undo the blocking after the pattern recognition phase.

```
for (j=1; j<=m; j++)
  EXCH_1( a[j][:] );  /* communication in dimension 1 */
  if ( owned_1( a[j][:] ) )
    code for MV(i,1, a[j][:], b[j][:], c[:][:], a[j][:]);
```

Note that this scheme already includes message vectorization. □

The implementations $\Delta[m]$, $\Xi[m]$, $\Psi[m]$ and $\Pi[m]$ are machine dependent and are parameterized in problem sizes and data distributions of the operand arrays occurring in the instance. They should be written in e.g. message–passing C with inline assembler to allow optimal usage of local processor features[4].

Note that a standard implementation may also result in some loop being executed sequentially if required by the given array distributions.

For some patterns m there may not exist a (non–standard) parallel algorithm. Furthermore, the user may a priori[5] forbid PARAMAT to select a parallel implementation different from the standard one for a specific pattern m by setting a flag bit NoReplace(m). To enforce a standard parallelization for the entire T_v, the NoReplace switch must be set for all the patterns matched at the nodes of T_v.

For an instance I of a pattern m, the boolean predicate NoParallel[m](I) evaluates to TRUE iff it is, given the problem sizes and data distributions, not advisable to generate parallel code for I. For this case, the effect is the same as setting ReplSeq[m].

For each pattern m, we build an *implementation driver* that generates code for any instance I of m. The coarse structure of such a driver looks as follows:

---

gen_code[m](I, T_v):
if SeqDebug[m] **then** generate $\Delta[m]$ for I; **return fi**;
if NoParallel[m](I) can be evaluated statically
then if NoParallel[m](I)
  then generate $\Xi[m]$ for I
  else if NoReplace[m] or there is no $\Pi[m]$
    then generate $\Psi[m]$ for $L - L'$ (see above) around *gen_code[r'.pat](r'.matched,* $T_{r'}$*)*
    else generate $\Pi[m]$ for I **fi**
  fi
else (some problem size is unknown at compile time)
  generate target code "`if` (NoParallel[m](I))";
  generate $\Xi[m]$ for I;
  generate target code "`else`";
  if NoReplace[m] or there is no $\Pi[m]$ available
  then generate $\Psi[m]$ for $L - L'$ (see above) around *gen_code[r'.pat](r'.matched,* $T_{r'}$*)*
  else generate $\Pi[m]$ for I
  fi
fi

Thus, if the problem size of I is known at compile time and if it is small, PARAMAT will decide to prohibit parallelization if sequential execution will be faster, thus avoiding slow-down of the target program. If the problem size is not known at compile time, a suitable run time test is inserted into the generated code.

Similar run time tests can be inserted if PARAMAT is not really sure about the value of certain important program values. An example is the following situation that is often encountered in multigrid applications: The programmer uses a large linear workspace array to store all (e.g., 2-dimensional) grids and indexes

[4]E.g. the arithmetic pipeline of the Intel *i*PSC/860 node processor *i*860 can only be used if the program is written in machine language — the C compiler does not vectorize. [Fri91] shows how impressive performance improvements can be reached by exploiting hardware features like arithmetic pipelines, dual operation mode or dual instruction mode that are just ignored by the standard compilers.

[5]This may also be handled by a compiler option included in the program text, but as we focus on fully automatic parallelization, this is not a viable alternative for us.

each single grid by using an offset pointer which is, in general, an array reference itself. Such indirect array accesses cannot be handled by compile–time data dependence analysis, and, even worse, a standard decomposition scheme for this linear workarray will result in bad load balancing and unnecessary communication. However, from the indexing schemes in recognized patterns of interpolation or restriction operations from one grid to the next one, PARAMAT is able to detect the (potentially) different grid parts by treating the offset array accesses as symbolic parameters. To make sure that the offset values implement the workspace concept, a suitable run time test on the offset values must be generated that, if successful, treats each single grid as a unique (2-dimensional) array that can be aligned and partitioned individually, thus avoiding the performance decrease mentioned above. As the number of different grids (and thus, the number of offsets) is usually small, this run time test does not involve much run time overhead. As the potential benefit from a positive test result is great, this optimization is sensible. If the assumption of a workspace grid hierarchy has been confirmed at run time, the workspace array is decomposed into the single grids, and program control branches to an alternative implementation with separate array distributions for each grid.

5.3 Examples for non–standard parallel implementations

This section gives some examples for parallel implementations that may differ completely from the original sequential program structure, or that introduce useful transformations of the corresponding standard implementation. The latter can be regarded as *automatic program transformation* which is hidden from the user. There is no need for a cyclic approximation scheme of successively applying some program optimizations, observing the results, and choosing better ones [HACZ93]. The disadvantage is that for each pattern a separate implementation driver is required. We claim that this can be taken into account, given that there would be a large intellectual effort devoted to the development of numerical software libraries for any real machine. In any case, we have finally the chance to get rid of the owner–computes–rule.

The implementations are code skeletons where the slot entries are entered in an appropriate way. They already contain message passing statements and register allocation. In the sequel, we sketch some of them. For a more complete survey of parallel algorithms for matrix computations, see [FJL+88] or [GHN+90].

Reduction operations For instances of specific common reduction operations (cf. Table 1) like global sum, global product, global OR, global maximum etc., we can make optimal use of optimized routines that are, in general, already supplied with the run time environment of the target machine. Here the non–standard parallel implementation mainly consists of a run time system call.

Grid relaxations A single grid relaxation step represents one update of all elements of a two-dimensional grid. A sequence of such steps, e.g. a step–counting loop around them, offers additional potential for optimizations.

Algorithm replacement must always be conservative with respect to numerical stability and convergency properties. As the recognized pattern's names are available, we can access mathematical background information, e.g., on convergency properties. This information allows — if not explicitly forbidden by the user — the replacement of, for instance, a Gauss-Seidel Wavefront relaxation by its Red-Black variant or by two steps of Jacobi relaxation which is much better suited for parallel execution (depending on the target machine). The basic motivation for this "aggressive" local replacement of implementations is that the average user just wants to get the actually fastest parallel implementation on *this* target machine — independent of, for instance, a particular relaxation coding.

Linear recurrences Simple linear recurrences are a classical example for algorithm replacement. Usually it appears as a sequential loop like

```
for (i=2;i<=n;i++) X[i]=(A[i]*X[i-1])+B[i];
```

which is serialized due to a loop carried data dependence as long as standard parallelization is used.

For recognized linear recurrences (here $\mathsf{FOLR}^{(1)}$) we can apply a suitable number of recursive doubling steps [KS73] to gain some parallelism while taking care of growing communication overhead. The optimal number of recursive doubling steps (up to $\max(p, \log n)$ are possible for p processors) depends on the problem size n and the time required for interprocessor communication on the target machine. For smaller problem sizes, the sequential variant will be faster.

Matrix-vector and matrix-matrix multiplication For matrix-vector multiplication ($\mathsf{MV}^{(2)}$), the standard method can be implemented as the `ij`–variant (the inner loop is a dot product) or as the `ji`–variant (the inner loop is a "`daxpy`" vector update). The latter variant seems to be preferable on vector node architectures.

Alternatively, we might use a systolic algorithm; this seems at most appropriate for transputer arrays with comparably low communication overhead and node performance.

For Matrix–Matrix Multiplication ($\mathsf{MM}^{(3)}$), the standard method expands to one of six possible variants (`ijk`, `ikj` etc.) since all three loops are interchangeable. An alternative would be a systolic implementation (see e.g. [FP92]). Similar systolic methods are also applicable to LU decomposition ($\mathsf{LUD}^{(3)}$).

Discussion Algorithm replacement must be conservative with respect to numerical stability and convergency properties of the recognized patterns. For each pattern m, the nonstandard implementation $\Pi[m]$ must guarantee that its numerical stability is not worse than that of $\Psi[m]$. Where this is not possible, the user receives a warning, and thus can force PARAMAT to choose the standard implementation by setting NoReplace[p].

Algorithm replacement is the most complex and strongest program transformation of all. Safe algorithm exchange is enabled only by the availability of pattern instances. It includes all other machine specific optimizing transformations. The implementation library can be optimized off-line by expert parallel programmers, until optimum performance is reached. Some optimizations may even be re–introduced which have been removed at the pattern recognition phase (e.g., loop blocking, semantically redundant `IF`s, etc.). The suitable communication routines, either simple SEND and RECEIVE instructions or higher–order communication primitives like COMBINE, REDUCE, BROADCAST, GATHER and SCATTER that are typically supplied with the parallel environment, are a basic component of the parallel pattern implementations and need not be further optimized afterwards. Such optimizations would usually be required for semiautomatically parallelized code, e.g. by vectorization of messages [Ger89], or by the general message passing optimization technique proposed in [LC91].

Algorithm replacement enables local deviation from the owner-computes-rule; it forms a framework to include all useful parallel algorithms that are known so far for the corresponding class of target machines (topology, granularity, communication properties). All expert's knowledge becomes available for the average user, although he/she does not need to be concerned about these algorithms or machine parameters.

5.4 Pattern-driven data distribution

To simplify the system design a given hardware environment is regarded as fixed; in particular, hardware resources like, the number p and the speed of the processors, the network topology, the cache size and caching strategy, and the memory size are regarded as constant. This corresponds to a 'dedicated' target machine. In the following, we need not consider these hardware parameters further. Nevertheless, scalability of parallel pattern implementations (in a more general sense) is still an important issue since local problem granularity still depends on the problem size.

Each parallel pattern implementation accesses data in an individual manner. Thus, for each pattern implementation, there is (at least) one favorite alignment (to minimize communication) and one favorite distribution (to maximize parallelism) of all the arrays for this pattern. The programmer knows these favorite alignment and distribution strategies for each pattern implementation. This information is stored in a table and can be accessed by the *data distribution driver* for each instance. Some examples of array alignment and distribution recommendations for standard parallel implementations are given in Table 4.

pattern	algorithm	align	distribute
MCOPY(A, B)	matrix copy	$A \equiv B$	arbitrarily
VCOPY(V, W)	vector copy	$V \equiv W$	arbitrarily
MJACOBI(A, B)	Jacobi step	$A \equiv B$	quadr. blocks
MM(C, A, B)	matrix multiply	$A \equiv C \ \vee \ B \equiv C$	A repl., B by col. or A by row, B repl.
VSUM(s, V)	vector sum	arbitrarily	arbitrarily
SSP(s, V, W)	dot product	$V \equiv W$	arbitrarily

Table 4: Array alignment and distribution recommendations for the standard parallelizations of some patterns.

A second requirement for on-line optimization of array distributions is that the parallel implementations are specified in a data-distribution-independent way. This may be technically arranged either by conditionals depending on the distribution parameters of one or several arrays, or by replication of parallel implementations, one for each possible distribution configuration. In each of these cases, it would be advisable to limit the possible distribution alternatives, instead of admitting arbitrary block-cyclic distributions of any block size. For vectors of length n, we allow the following distributions:

(1) contiguous distribution (block size is n/p),
(2) cyclic distribution (block size is 1) and
(3) total replication (no distribution).

For a $m \times n$ matrix, we admit the following distributions:

(1) contiguous row distribution (block size is mn/p, block shape is $(m/p) \times n$),
(2) contiguous column distribution (block size is mn/p, block shape is $m \times (n/p)$),
(3) cyclic row distribution (block shape is $m \times 1$),
(4) cyclic column distribution (block shape is $1 \times n$),
(5) contiguous quadratic blocks (block size is mn/p, block shape is $(m/\sqrt{p}) \times (n/\sqrt{p})$) and
(6) total replication (no distribution, block shape is $m \times n$).

This limitation of array distribution alternatives is supported by the fact that for all our patterns [Keß94a], a locally optimal distribution for each array operand is contained in this list. We are aware of the fact that a *globally* optimal data distribution configuration may be made up of only locally suboptimal array distributions, although we believe that this scenario hardly appears in practice.

Quadratic contiguous block distributions are optimal for grid relaxation sweeps, since they minimize the surface-to-volume–ratio of the array partitions and thus the amount of data to be exchanged. In our framework, they are the only distribution scheme that distributes processors along more than one array axis. For quadratic distributions, however, we must add in this case the following constraint: The array (grid) A accessed by a matrix m must be 2–dimensional. Otherwise, imagine the following situation: Let A be three–dimensional, with axes A_1, A_2 and A_3, being distributed into quadratic blocks along, say, axes A_2 and A_3. Let m be a matrix access along the first and second axis of A. The number of processors along axis A_2 is $\sqrt{p}$, the number of processors along axis A_1 is 1 (not distributed). Thus, m has only $\sqrt{p}$ partitions, which limits parallelism unnecessarily, and, worse still, the overall number of working processors is no longer constant for each call to the corresponding relaxation routine. Since we do not want to do everything twice, with one extra routine version for p and one for only $\sqrt{p}$ processors, we generally admit quadratic block distributions only for arrays of dimensionality equal to 2.

The alignment and distribution recommendations for different pattern instances in a given program will usually conflict with each other. The problem of resolving this conflict by determining globally optimal data alignment and distribution is well–known to be NP–complete [LC90], thus automatic partitioning may take exponential time in the worst case. [DHR94] proposes a branch-and-bound algorithm for automatic partitioning. To help with the combinatorial complexity, we make use of our knowledge on favorite local partitionings as starting configurations when performing a global search for the optimal data distribution.

[DHR94] also covers *static array redistribution* which is a NP–complete problem itself [Kre93]. The main problem in static redistribution is that a globally optimal distribution scheme involving redistribution may even be made up of suboptimal data distributions for *all* phases of the program. However, [BKK93] shows that for application programs of moderate size (800 lines) represented as a sequence of phases, an *optimal* data distribution scheme can be found within a few CPU seconds using a fast 0-1 integer programming tool. This method matches our approach well, since the pattern instances supply the required phase representation, and the run time tables (see the next section) deliver suitably accurate cost estimates.

5.5 Pattern-driven run time prediction

Many performance prediction approaches [Gup92, Fah93, DHR94] work *analytically* by estimating the program's run time bottom-up through the abstract syntax tree, starting at the leaves of the expression trees, with an idealized model of the target machine in mind. Specific hardware features, like e.g. caches or network traffic, yield actual run times that significantly differ from the prediction. For this reason, we follow a *synthetic* performance prediction approach that has been proposed in [BFKK91] and [FMPB94].

For each pattern implementation, PARAMAT provides a *run time prediction driver* that inspects a table of previously measured run times of that implementation with varying problem sizes and varying array distribution schemes on the target machine. The table entries for each pattern are indexed in different data distribution configurations, in the problem sizes (logarithmic scale), and in the NoReplace flag. They also depend on the NoParallel predicate. The restriction of data distribution alternatives given above keeps the table sizes moderate. In addition, we require some table entries for the communication routines that may be generated due to array redistribution, see [BKK93].

As a consequence, run time prediction considerably gains accuracy since now actual run times of high-level implementations on the target architecture are available which reflect hardware properties (traffic on the network, message buffer sizes, message protocols, undocumented communication behaviour, overlapping of computation and communication etc.) better than theoretical, idealized estimation functions.

This synthetic run time prediction has another important advantage over the analytical approaches: it is faster, because table lookup suffices where otherwise complex intermediate representations have to be traversed and analyzed. For instance, the Addap[DHR94] system's automatic data distribution engine suffers mainly from slow analytical performance estimation.

Problems with performance prediction generally arise if the target machine has a cache. Then, run time also depends on whether operands (arrays or parts thereof) already reside in the cache due to a previous operation, or whether must be reloaded first. This scenario may be influenced by previous operations. With a synthetic approach, however, the larger the problem sizes are, the less this effect changes the actual run times compared with the table entries. For small problem sizes, the run time prediction drivers may be augmented by some correction term addressing the cache effect. This issue is left to future research.

Problem sizes (corresponding to vector lengths or matrix extents) need to be considered only in a specific interval $[N_{min}...N_{max}]$ of interest, e.g., from 8 to 16384. The parameter extent of that problem size axis thus contains $D = \log N_{max} - \log N_{min} + 1$ entries.

With these guidelines and with the limitation of array distribution alternatives given in the previous section, the parametrization space (and thus, the run time table size) for a pattern implementation with x vector operands, y matrix operands and z problem sizes contains $3^x \cdot 6^y \cdot D^z$ entries. For the MV matrix vector product, we obtain an (uncompressed) table size of $54D^2$. Of course, this does not mean that we have to implement matrix vector product once for each of these configurations. Generally, several entries can be handled as a whole block, e.g., by taking array alignment relations [LC90, KLS90, KN90] into account, or ranges of problem sizes with similar run time behaviour. The run time tables can also be compressed according to this hierarchical parametrization structure of the parallel implementation.

For run time prediction, we consider a parallel implementation ($\Psi[m]$ and $\Pi[m]$) of a pattern m as a *black box*. We are not concerned with the issue of how their run times *should* behave in theory, but how they actually behave on the concrete hardware configuration, which can substantially differ from the former.

The synthetic performance prediction treats greater code portions as units where analytical methods estimate the program's run time bottom up, starting at the expression level. Synthetic prediction models (at least partially) the cache behaviour due to the locality relations that are inherent to the parallel implementation, the overlapping of computation and communication, and the characteristic network traffic induced by the access structures inherent to the parallel implementation.

For true parallel algorithms ($\Pi[m]$) the analytical methods (like [Gup92, Fah93, DHR94]) often fail because they rely on standard parallelizations within a specific compilation environment. Synthetic performance prediction works also for all nonstandard parallelizations.

As a byproduct, the run time tables will provide an extensive performance spectrum of the target machine. Furthermore, it will show which parallel algorithms are feasible in practice, and in which range of problem sizes and for which data distributions they are superior to others or to standard implementations.

5.6 System overview

There remains the technical problem of how to code a parallel implementation in a data-distribution-independent way while maintaining explicit formulae for iteration and communication sets, and avoiding the overhead involved in evaluating complicated parametrization formulae at the target program's run time. We do this in two steps. First, PARAMAT specifies the parallel implementation in a target-machine-specific language like C plus inline-assembler. This specification however allows complicated parametrization formulae or, if unavoidable, excessive replication of implementation code. Once the data distribution engine has determined a global distribution configuration for all array operands, we can derive the proper parallel implementation subroutines (comparable to those in the previous section) from that data-distribution-independent specification by partial evaluation (for reference, see [JGS93]) and dead code elimination. We obtain small and efficient message-passing–C sources that are data-distribution-dependent, and we need to extract only those routines from the specification library that are called by the matched user program. These are then compiled and linked together with the matched user program that has been produced by a suitable code driver (cf. Figure 2).

These routines extracted from the specification are also used to produce the run time tables. As this is a tedious procedure, we plan to automatize table construction.

Note that the time–consuming generation of the run time tables can be performed off-line (at compiler generation time). We intend to develop an automatic benchmarking tool that does this tedious job.

For non-recognized code portions, PARAMAT generates standard parallelizations. The difference from standard parallelizations of recognized code portions is only that there are no corresponding entries in the data distribution/alignment recommendation and run time tables available; thus these code portions do not (yet) influence the global determination of array distributions.

6 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, vectorization or parallelization: Snyder [Sny82] 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. — [BS87] 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 [Bos88b, Bos88a] is an expert system for interactive vectorization of FORTRAN programs. It contains a simple pattern matching tool that can discover order 1 patterns (vector operations, reductions). — The pattern matcher of [PP91] works on a modified program dependence graph (PDG, see [FOW87]) 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

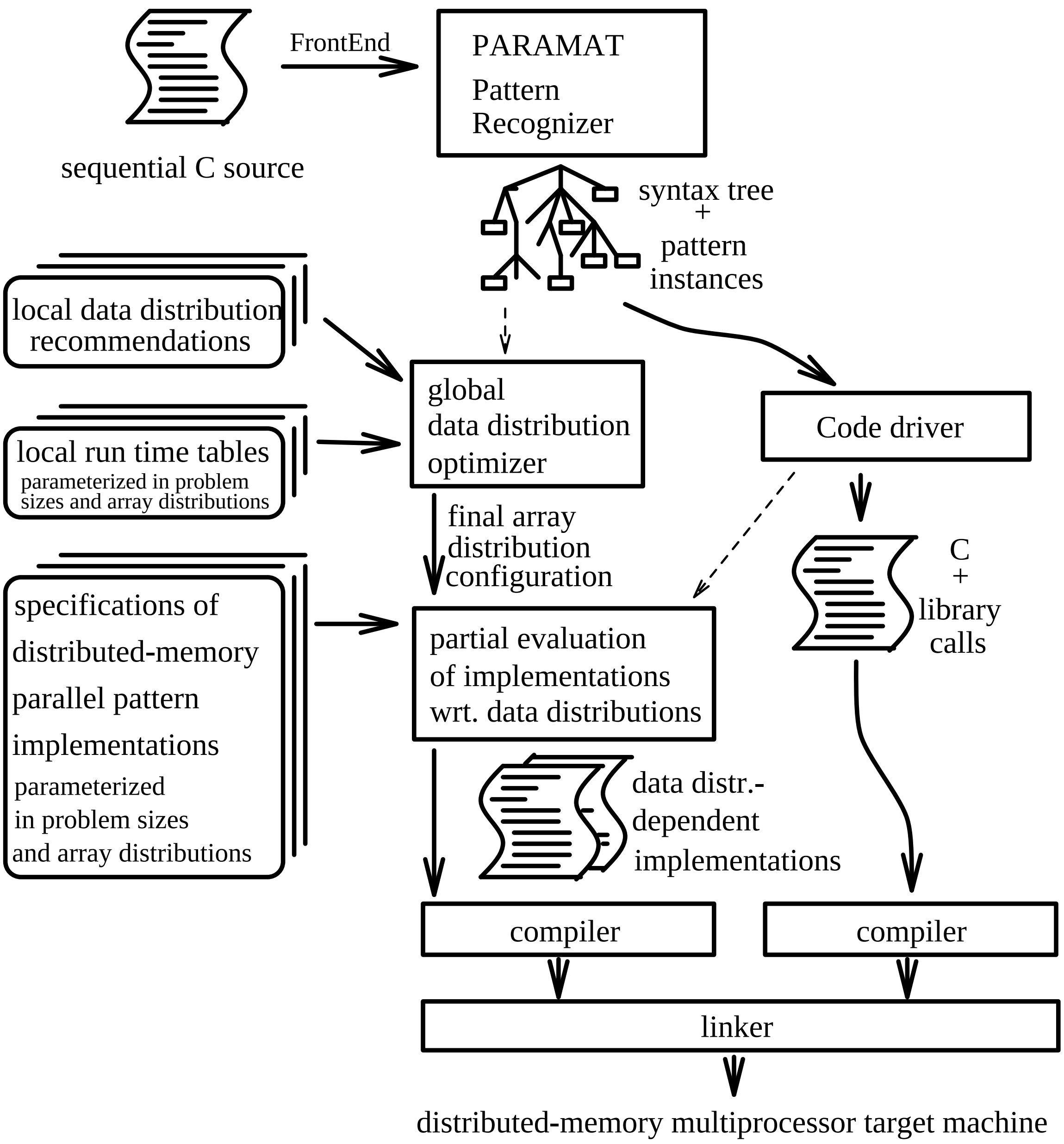


Figure 2: The overall structure of a distributed-memory back-end for PARAMAT.

higher because the rewrite rules form a graph grammar. Normalization of the PDG has to be provided interactively by the user. — By abstract interpretation of the sequential source, [AH90] computes a sequential memory access map (*abstract store*) that assigns to each array element referenced in a loop the corresponding symbolic representation of its content. Thereafter, loops are, where possible, replaced by their explicit representation (*closed form*), comparable to our pattern instances. They recognize some patterns of order ≤ 1, namely equivalents of POWER, VSUM$^{(1)}$, VPROD$^{(1)}$, PREVSUM$^{(1)}$, SSP$^{(1)}$. Based on the closed forms, they implemented recognition of induction variables. The method fails at unrolled or blocked loops. — [RF93] 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 unjustified. — CMAX [SW93] 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 in order to make the program machine-independent and more suitable to distribution of data at that point.

Program comprehension for algorithm replacement should not be confused with pattern matching that optimizes communication statements, e.g. in [IFKF90] and [LC91]. These approaches do not try to understand program semantics but apply pattern matching to (implicit) message–passing code to exploit higher–order communication routines like global combine, reduction, or broadcast, that are supplied by most parallel run time systems. Note that such optimizations are contained in PARAMAT's algorithm replacement strategy.

Other current research projects: [BHRS94] concludes, from a case analysis, that current tools for automatic parallelization are not powerful enough and recommend pattern recognition as the solution. Some general ideas are sketched, but there is no implementation. — [DI94] builds from the PDG a database of PROLOG facts, formulates templates as PROLOG clauses and uses PROLOG's inference engine for pattern matching. This approach, although slightly more general than ours, forbids intermediate restructuring, relies on backtracking and takes exponential run time in the worst case. The information derived is used in an interactive system for automatic array alignment and distribution [HACZ93]; algorithm replacement is not straightforward as in PARAMAT. A detailed comparison of this approach with PARAMAT's pattern recognizer is given in [dK96]. — A program comprehension system for FORTRAN programs sketched in [Met95] is currently being implemented for a list of over 500 idioms of common loop nests, which corresponds roughly to an uncompressed version of our PHG. The method works on the PDG; it is a top–down approach that partly uses the algorithm from [Sny82].

Other problem domains: Some systems for program comprehension in a non–numerical domain are targeted towards automatic documentation and support of software maintenance. Transformation or replacement of code is not considered. Plan Calculus [RW90] 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. — The PAT approach [HN90] and following work [KNE93] uses an abstract, object-oriented representation for syntactic and semantic concepts composing a (COBOL) source 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.

7 Conclusion

The PARAMAT approach to automatic parallelization consists of three basic ideas: First, we observe that we can cover large parts of many numerical codes by a small set of typical programming patterns. Second, we devise a recognition algorithm similar to bottom-up pattern matching which tries to locally recover the semantics of the program, while being robust against many common code modifications such as loop distribution, loop interchange, loop blocking or loop unrolling. Third, we use the restored program semantics information to guide sophisticated optimizing code transformations including local algorithm replacement.

In this paper, we have presented a powerful framework for the detection of the patterns in scientific programs. We applied our knowledge on the semantical correlations between the patterns for speed and space economy. We used data access description and data flow information to compute cross edges which guide recognition of delocalized code portions. Our prototype implementation shows (1) that pattern recognition is robust against many common code transformations, (2) that writing code for template realizations is rather easy, and (3) that pattern recognition is very fast.

We have presented a framework for pattern–driven generation of parallel code. For each pattern we can — as an alternative to standard parallelization of some loops according to given array distributions — also select a conceptually different parallel algorithm, for instance, highly optimized system routines supplied with the hardware environment. Safe algorithm replacement, though, is only guaranteed by the availability of pattern instances. It provides a universal framework to integrate all known parallel algorithms, library routines, and program transformations. Treating larger code parts as atomic building blocks of a parallel program also supports faster and more accurate performance prediction. Thus, PARAMAT makes the experience of parallel programming and optimization experts accessible to all scientific programmers and thus avoids re-inventing the wheel for each program parallelization project.

PARAMAT is *not* interactive. This is not necessary either because the user does not have to recognize his/her code during and after parallelization for selecting transformations or further tuning by hand. On the other hand, this 'non-WYSIWYG' system offers many more possibilities for aggressive optimizations and hides the parallelization details from the user.

The PARAMAT system is open for extensions. The pattern library can be extended by adding more pattern modules according to individual application areas. The computation of the run time approximation functions can be automatized by a universal benchmarking tool. Changing the hardware platform only requires the loading of another base of parallel implementations, their default distributions, and their run time functions. Thus the PARAMAT system can always be up to date with the latest available hardware environments.

The PARAMAT system could also be modified to output HPF source programs instead of target machine code. As HPF programs (especially, distribution and mapping directives and explicitly transformed code) are target-machine (and compiler-) specific, generating HPF output for each pattern by the implementation drivers and distribution recommendations by the distribution drivers is, in principle, possible. This, however, would only work if the same HPF target compiler is used to generate the machine code, since this compiler must then also be used to generate the run time tables for the pattern implementations written in HPF. On the one hand, this would supply a Fortran 77 (Fortran 90, C) to HPF converter for a specific target machine; on the other hand, it is likely that this indirect approach of generating HPF code and later compiling it again will result in a performance degradation of the final target program, compared with direct machine code generation by PARAMAT.

Acknowledgements

We gratefully acknowledge fruitful discussions with Prof. Dr. Wolfgang Paul, Prof. Dr. Helmut Seidl, and Prof. Dr. Reinhard Wilhelm.

References

[AH90] Zahira Ammarguellat and W.L. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *ACM SIGPLAN Programming Language Design and Implementation*, 1990.

[BBC+93] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *TEMPLATES for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1993.

[Ber92] Michael Berry (Editor). Scientific Workload Characterization by Loop Based Analyses. *Performance Evaluation Review*, 19:17–29, February 1992.

[BFKK91] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, volume 3, pages 213–223, 1991.

[BHRS94] S. Bhansali, J.R. Hagemeister, C.S. Raghavendra, and H. Sivaraman. Parallelizing sequential programs by algorithm-level transformations. In *Proc. Third Workshop on Program Recognition*, pages 100–107. IEEE Computer Society Press, 1994.

[BKK93] Robert Bixby, Ken Kennedy, and Ulrich Kremer. Automatic Data Layout Using 0–1 Integer Programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, Houston, TX, Nov. 1993.

[Bos88a] Pradip Bose. Heuristic Rule-Based Program Transformations for Enhanced Vectorization. In *Proc. of Int. Conf. on Parallel Processing*, 1988.

[Bos88b] Pradip Bose. Interactive Program Improvement via EAVE: An Expert Adviser for Vectorization. In *Proc. Int. Conf. on Supercomputing*, pages 119–130, July 1988.

[BS87] Thomas Brandes and Manfred Sommer. A Knowledge-Based Parallelization Tool in a Programming Environment. In *16th Int. Conf. on Parallel Processing*, pages 446–448, 1987.

[CK88] David Callahan and Ken Kennedy. Compiling Programs for Distributed Memory Multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[CMZ92] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[DDHH88] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. on Math. Software*, 14(1):1–32, 1988.

[DHR94] Anne Dierstein, Roman Hayer, and Thomas Rauber. Automatic parallelization for distributed memory multiprocessors. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 192–217. Verlag Vieweg, 1994.

[DI94] B. DiMartino and G. Ianello. Towards Automated Code Parallelization through Program Comprehension. In *Proc. Third Workshop on Program Recognition*, pages 108–115. IEEE Computer Society Press, 1994.

[dK96] Beniamino diMartino and Christoph W. Keßler. Program comprehension engines for automatic parallelization: A comparative study. In *to appear in: Proc. First Int. Workshop on Software Engineering for Parallel and Distributed Systems*. Chapman&Hall, March 25–26 1996.

[Fah93] Thomas Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. PhD thesis, Technisch-Naturwissenschaftliche Fakultät der Universität Wien, 1993.

[Fea91] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *Int. Journal of Parallel Programming*, 20(1):23–53, Feb. 1991.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, vol. 1: General Techniques and Regular Problems*. Prentice–Hall, 1988.

[FMPB94] Arno Formella, Silvia Müller, Wolfgang Paul, and Anke Bingert. Isolating the Reasons for the Performance of Parallel Machines on Numerical Programs. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 34–64. Verlag Vieweg, 1994.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FP92] T.L. Freeman and C. Philips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

[Fri91] Stephen S. Fried. Personal Supercomputing with the Intel i860. *BYTE*, pages 347–357, Jan. 1991.

[FSW92] Christian Ferdinand, Helmut Seidl, and Reinhard Wilhelm. Tree Automata for Code Selection. In *Code Generation – Concepts, Tools, Techniques*, pages 30–50, 1992.

[Ger89] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, Universität Bonn, 1989.

[GHN+90] K.A. Gallivan, M.T. Heath, E. Ng, J.M. Ortega, B.W. Peyton, R.J. Plemmons, C.H. Romine, A.H. Sameh, and R.G. Voigt. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, 1990.

[Gup92] Manish Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Technical Report UILU-ENG-92-2237 or CRHC-92-19, 1992.

[HACZ93] Jan Hulman, Stefan Andel, Barbara M. Chapman, and Hans P. Zima. Intelligent Parallelization within the Vienna Fortran Compilation System. In *Fourth Workshop on Compilers for Parallel Computers*, pages 455–467, Dec. 1993.

[HKT91] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler–Support for Machine–Independent Parallel Programming in Fortran-D. Technical Report Rice COMP TR91-149, Rice University, March 1991.

[HN90] Mehdi T. Harandi and Jim Q. Ning. Knowledge-based program analysis. *IEEE Software*, pages 74–81, January 1990.

[HPF93] High Performance Fortran Forum HPFF. High Performance Fortran Language Specification. *Scientific Programming*, 2, 1993.

[IFKF90] K. Ikudome, G. C. Fox, A. Kolawa, and J. W. Flower. An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers. In *Fifth Distr. Memory Computing Conf. (DMCC5), IEEE Computer Society Press*, pages 1105–1114, 1990.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[Keß94a] Christoph W. Keßler. *Automatische Parallelisierung numerischer Programme durch Mustererkennung.* PhD thesis, Universität Saarbrücken, 1994.

[Keß94b] Christoph W. Keßler. Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. In *Proceedings of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems.* Birkhäuser Verlag AG, Basel, Switzerland, April 1994.

[KLS90] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[KN90] Kathleen Knobe and Venkataraman Natarajan. Data Optimization: Minimizing Residual Interprocessor Data Motion on SIMD machines. In *Third Symposium on the Frontiers of Massively Parallel Computation*, pages 416–423, 1990.

[KNE93] Wojtek Kozaczynski, Jim Ning, and Andre Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12), Dec. 1993.

[Kre93] Ulrich Kremer. NP-Completeness of Dynamic Remapping. Technical Report CRPC-TR93330-S, Center for Research on Parallel Computation, Rice University, Houston, TX, Aug. 1993. see also: Proc. Fourth Workshop on Compilers for Parallel Computers, Delft, Dec. 1993.

[KS73] Peter M. Kogge and Harold S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

[LC90] Jingke Li and Marina Chen. Index Domain Alignment: Minimizing Cost of Cross–referencing between Distributed Arrays. In *Third Symposium on the Frontiers of Massively Parallel Computation*, pages 424–433, 1990.

[LC91] Jingke Li and Marina Chen. Compiling Communication–Efficient Programs for Massively Parallel Machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):361–375, July 1991.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. on Math. Software*, 5:308–325, 1979.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *ACM SIGPLAN Principles of Programming Languages*, Jan. 1993.

[McM86] Frank McMahon. The Livermore Fortran Kernels: A Test of the Numeric Performance Range. Technical report, Lawrence Livermore National Laboratory, 1986.

[Met95] Robert Metzger. Automated Recognition of Parallel Algorithms in Scientific Applications. In *Workshop on Plan Recognition at IJCAI'95*, Aug. 1995.

[MFL+92] A. Mohamed, G. Fox, G. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Application Benchmark Set for Fortran D and High Performance Fortran. Technical Report 327, Northeast Parallel Architectures Center, 1992.

[PP91] Shlomit S. Pinter and Ron Y. Pinter. Program Optimization and Parallelization Using Idioms. In *ACM SIGPLAN Principles of Programming Languages*, pages 79–92, 1991.

[PTVF92] William H. Press, Saul A. Teukolski, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C — The Art of Scientific Computing, second edition.* Cambridge University Press, 1992.

[RF93] Xavier Redon and Paul Feautrier. Detection of Recurrences in Sequential Programs with Loops. In *PARLE 93, Springer LNCS vol. 694*, pages 132–145, 1993.

[RW90] Charles Rich and Linda M. Wills. Recognizing a Program's Design: A Graph–Parsing Approach. *IEEE Software*, pages 82–89, Jan. 1990.

[SLY90] Z. Shen, Z. Li, and P. Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. *IEEE Trans. Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[Sny82] Lawrence Snyder. Recognition and Selection of Idioms for Code Optimization. *Acta Informatica*, 17:327–348, 1982.

[SW93] Gary Sabot and Skef Wholey. Cmax: a Fortran Translator for the Connection Machine System. In *Int. ACM Conference on Supercomputing*, pages 147–156, 1993.

[WS90] Debbie Whitfield and Mary Lou Soffa. An Approach to Ordering Optimizing Transformations. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 137–146, 1990.

[ZBG88] Hans Zima, Heinz Bast, and Michael Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison–Wesley, 1990.