8 Knowledge–Based Automatic Parallelization by Pattern Recognition

Christoph W. Keßler
GRADUIERTENKOLLEG INFORMATIK
SAARBRÜCKEN UNIVERSITY, GERMANY
email: kessler@cs.uni-sb.de

Abstract: We present the top–down design of a new system which performs automatic parallelization of numerical Fortran77, Fortran90 or C source programs for execution on distributed–memory message–passing multiprocessors such as e.g. the INTEL iPSC/860 or the TMC CM–5.

The key idea is a high–level pattern matching approach which in some useful way permits partial reverse–engineering of a wide class of numerical programs. With only a few hundred patterns, we will be able to completely match many important numerical algorithms. This is also applicable to so-called dusty deck sources that may be 'encrypted' by various former machine-specific optimizations.

We show how successful pattern matching enables safe algorithm replacement and allows more exact prediction of the performance of the parallelized target code than usually possible. Together with mathematical background knowledge and parallel compiler engineering, this opens access to a new potential for automatic parallelization that has never been exploited before.

8.1 Introduction and Overview

Current distributed memory multiprocessers are hard to program. Predicting the performance of a nontrivial parallel program is not easy either. Thus it is a natural consequence to leave as much as possible of this – often tedious – work to an optimizing parallelizing compiler. The programmer just wants to feed in the sequential program and get out optimized parallel code.

Today, truly automatic parallelization is yet a dream (and some people believe it will remain so forever). Currently, research on automatic methods appears rather at the border of compilers for parallel programming languages for distributed memory systems, e.g. [7] for Fortran D ([20]) or [10, 11] for Vienna Fortran 90 ([9]).

The state of the art in both parallelizing compilers and compilers for parallel programming languages for distributed memory multiprocessors is semiautomatic parallelization: The programmer supplies the compiler with array distribution specifications (e.g. interactively, as in SUPERB ([15]) or in the form of compiler directives, as in Fortran D), and, generally, transforms the program himself in order to efficiently make use of machine–specific properties (e.g., buffer sizes, message protocols, processor distances and so on). The job of the compiler is to insert masks and communication statements according to the distribution specifications. Automatic optimization of masks and communication is possible in a limited way (see [15]).

The semi-automatic approach enables the programmer to write his code in the single–program–multiple–data programming paradigm which is well suited for scientific (numeric) applications. However, the efficiency of the generated parallel code is heavily dependent on the chosen data distribution.

This user interaction seems inevitable since the problem of determining optimal array distributions is a hard one, and only the user seems to possess sufficient knowledge about his program in order to decide which distributions to choose and how to transform his code to maintain parallel program efficiency.

However, for large application programs with lots of arrays and complex distribution relations between them, this becomes impracticable because the user cannot solve this optimization problem either. Moreover, detailed knowledge of the target machine must be available at this step, and different parts of the application program may require different data partitioning schemes to perform efficiently. Where may redistribution be useful, and where not? Furthermore, the quality of a given data distribution is also dependent on the underlying hardware. There is no automatic guidance in choosing suitable code transformations for better utilization of hardware properties. Changing the hardware platform means also repeating the whole parallelization procedure. Should an average user — who is, in general, not an expert in parallel computing — really care about all this stuff?

Recently some research has been done on automatic derivation of data distribution ([24, 17, 29, 37, 21]), but we are missing a really automatic approach which leaves no partial work to the programmer any more and requires no user interaction.

Some recent research has recommended to apply pattern recognition techniques on a quite low level to facilitate parallelization and optimizations ([17, 33, 10, 11]). We claim that pattern matching can be usefully applied to entire programs as the *core* of an automatic parallelization system. Only high–level pattern recognition will enable knowledge-based program transformations by decrypting the meaning of the program which is often hidden past recognition within the source code by former target–specific transformations and programming styles.

Earlier approaches have suggested to apply program concept recognition to guide vectorization ([5, 4]) and parallelization ([6, 36, 11]) but have not been developed further. The PAT system performs concept recognition for non-numerical codes (see [18, 27]). However, pattern recognition techniques have never been applied on a large scale in a 'fully automatic' parallelization system.

How can pattern recognition techniques be motivated? In order to answer this question, we have examined lots of numerical application algorithms that are typically run on distributed memory multiprocessors, e.g. the algorithms considered in [32] or the algorithms occurring in a parallel numerics course [30]. These algorithms contain basic linear algebra subroutines (see also [28, 12]), direct 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 differential equations), interpolation problems, numerical integration and differentiation, and multigrid algorithms[1].

[1] We have focussed in this work on algorithms operating on rectangular dense real matrices. Our approach may easily be extended to other matrix types (e.g., banded, blocked, triangular; complex).

We observed that these numerical algorithms are made up of only a few (around 100) characteristic programming schemes (called *patterns*) such as e.g. vector and matrix operations, simple recurrences, relaxation operations or simple reduction operations. We describe these patterns in more detail in section 8.3.

We claim that these few patterns cover a broad range of numerical application codes that are actually run on distributed–memory multiprocessors. We exemplify this in section 8.3 by examining the source codes of actual application programs.

Faced with these real–world codes, the pattern matcher must be robust against semantics preserving code transformations, in order to maintain acceptability. In general, there are several different possibilities (called *templates*) to implement a specific pattern. Thus a pattern is, in some way, a normal form of all its templates. The job of pattern matching is to compare a given piece of the source program with the templates, to choose the corresponding pattern and to replace this program piece by an instance of that pattern. We will describe our pattern matching algorithm in detail in section 8.4.

Once this pattern recognition tool performs well, the rest is quite simple: locally find out what the programmer's intention was, and then select well–suited and highly optimized target code for this piece of the application. If necessary, these code pieces must be connected by appropriate redistribution operations.

The remaining sections describe the other components of our parallelization system called PARAMAT ("PARallelize Automatically by pattern MATching") that use the pattern matcher's output in order to generate efficient parallel code.

8.2 Preprocessing the Source Code

It is very important that the program is rather explicit when it is submitted to automatic parallelization by pattern matching. Beyond a sophisticated dependence analysis, a pass of preparing code transformations[2] should be carried out in order to facilitate pattern recognition. Often, semantics preserving code modifications had been applied to Fortran codes to optimize sequential or pipelined execution. However, this made the code less readable, and for pattern recognition, they impose unnecessary barriers which must be removed[3] by the following transformations:

1. **procedure inlining**[4]
2. **constant propagation**

   Example: Consider the following situation:

[2] For a detailed description of optimizing transformations, see e.g. [39].

[3] We remind that such optimizations may be re-inserted later on at code generation time – well-tailored to the parallel target machine.

[4] This is only for a prototype version of PARAMAT in order to make the implementation easier. Interprocedural analysis is currently evolving, and we will include interprocedural analysis tools into the final system as soon as they become adequately reliable. Our case studies have shown that for purely numerical programs, procedure inlining will blow up program size by only a small constant factor. Since this only appears at compile time, it can be tolerated.

```
N = 1024
.....
NP1 = N + 1
.....
```

Such constructs are widely used to enforce the compiler to use registers or to avoid recomputation of often–needed common constant subexpressions. Here, `N + 1` should be replaced by its actual value 1025. This propagates information about the constant values into the 'interior' of the program where it is needed at the pattern recognition phase.

3. **induction variable recognition** (substitution, if possible)

   Induction variables are integer variables used to abbreviate array indexing expressions. They often are introduced into program codes to optimize address calculation time and to enforce the compiler to use address registers for these expressions.

   Example: Consider the following code fragment:

```
   DO 20 J=2,NC
    JF=J+J
    DO 10 I=2,NC
     IF1=I+I
     FC(I,J)=2*(FF(IF1,JF)-4.0*UF(IF1,JF)+UF(IF1,JF-1)
  *            + UF(IF1-1,JF)+UF(IF1+1,JF)+UF(IF1,JF+1))
10  CONTINUE
20 CONTINUE
```

   The introduction of the induction variables `JF` and `IF1` saves many integer multiplications by 2 but hides important information from the pattern recognition phase. For this reason, the induction variables must be substituted away by propagating their values `2*J` and `2*I`, respectively.

4. **temporary variable recognition** (substitution if possible)

   Temporary variables have often be inserted into a program only to denote common subexpressions, to enforce use of a data register for that variable or just to make the code more readable. In such a case, the variable may be substituted away.

   Example: Consider the following matrix multiplication code:

```
do i=1,64
 do j=1,64
  temp(i,j) = 0.0
  do k=1,64
   temp = temp + a(i,k) * b(k,j)
  enddo
  c(i,j) = temp
 enddo
enddo
```

The variable `temp` can be recognized as temporary and substituted away by `c(i,j)` since the value of `temp` is not used further.

5. **dead code elimination**

   Example: In the example above, the assignment to `c(i,j)` changes into `c(i,j) = c(i,j)` which is redundant and can be removed.

6. **conversion of GOTO's** into if–then–else or while statements, where possible.

   This feature concerning especially old (Fortran77) codes is important since GOTO's cannot be handled by the current version of the pattern recognition tool.

7. **IF-outermosting**

   An IF statement checking a condition not depending on the index of the surrounding loop (after induction variable detection!) should be moved before this loop.

8. **array simplification**

   Each array A containing a dimension d with an extent e_d^A smaller than 16 will be splitted into e_d^A different arrays $A1, \ldots, Ad$ of dimensionality $d-1$ each.[5]

9. **loop distribution**

   If not prevented by data dependency cycles, this transformation supplies the pattern recognition phase with handy, small, mostly perfectly nested loops.

   Example:

```
do i=1,128                        do i=1,128
 do j=1,256                         do j=1,256
  c(i,j)=b(j)*c(i,j)     ---->        c(i,j)=b(j)*c(i,j)
 enddo                              enddo
 a(i)=c(i,1)                      enddo
enddo                             do i=1,128
                                    a(i)=c(i,1)
                                  enddo
```

Furthermore we disallow constructs causing run–time dependencies which cannot be recognized by the prototype version of PARAMAT. This especially concerns index vectors, so programs containing indirect array references will be rejected just at the beginning. For these cases, dynamic techniques must be applied (see e.g. [34]). In this work we restrict ourselves to static parallelization.

[5]Loops stepping through dimension d of A must then be completely unrolled (i.e., replicated $E_d^A - 1$ times) and the Aj must be inserted instead of the $A[j]$. If there arise compile–time–unknown factors in A's indexing, or if loops step through this dimension in an irregular manner, this transformation will not be possible.

8.3 Which Patterns are Supported?

Now, let us describe which patterns should be included into the pattern library of PARAMAT. On the one hand, we want to cover a very high percentage of numerical programs, on the other hand we must not use too many patterns, leading to inacceptable compile times.

The basic algorithms considered in [32] and [30] suggest that a rather small number of patterns will suffice to cover large parts — especially the time–critical ones — of real application programs. In order to exemplify this assumption, we took a closer look to some real–world codes (after being normalized by the transformations described in the last section):

- the Livermore Loops (cf. [31]),
- some kernels from the NAS Benchmark program (cf. [1]),
- a LU decomposition code from the netlib,
- a least–square Conjugate Gradient solver[6] from [30],
- two multigrid programs,
- selected codes from the Perfect Club Benchmarks (especially from the programs FLO52, SPEC77 and DYFESM, see also [3]).
- and others.

Faced with these codes, we created appropriate patterns, subpatterns and templates while carefully making the patterns robust against many possible semantics–preserving code modifications. The result of this research, the current version of the Basic PARAMAT Library, unfortunately cannot presented here for lack of space, but it is listed in [23]. A brief summary of the patterns is given in Tab. 8.1. Some ideas for the efficient distributed–memory parallel implementation of all the patterns occurring in these algorithms are summarized in [32].

As one can see from Tab. 8.1, the number of low–level patterns (expression and statement level) is rather limited; so is the number of medium–level patterns (loop level, e.g. vector instructions). The number of medium–level patterns can additionally be restricted by maintaining loop normal forms generated by loop distribution; we will discuss this following the example of section 8.4.4. The most critical point is how many *high*–level patterns (and which) to include in the library; they are often too specific to include them into the Basic Library. This question will later be alleviated by introducing a modular concept where the pattern library may be individually composed from the basic and other more specific sub–libraries in a hierarchically organized database.

With the current version of the Basic Pattern Library — containing only 120 patterns — we are able to cover *completely* (and thus, to parallelize automatically) a lot of the

[6] E.g. the main patterns generally contained in CG–algorithms are CGINIT$^{(1)}$, MV$^{(2)}$ or VM$^{(2)}$ (vector–matrix multiply), VADD$^{(1)}$, VADDMUL$^{(1)}$ (vector instructions), SSP$^{(1)}$(standard scalar product) and SV$^{(1)}$ (scalar–vector multiply).

order	patterns	number
0	scalar arithm. operations, init, max, min, swap, assign, read write	20
	MULTIADD/-MUL, difference stars and substars (first and second order)	5
1	vector instructions, v-init/-assign/-copy/-swap, v-read/write, SV	16
	reductions: v-sum, scalarproducts, vector norms	8
	reductions: vector maximization/minimizations	6
	1D relaxation steps (Jacobi, Gauss–Seidel)	2
	first order linear recurrences	2
	intermediate forms of 1D convolution	2
2	matrix operations, m-init/-assign/-copy, m-read/-write, SM	12
	vector/matrix multiplication patterns	4
	2D-reductions: m-sum, concurrent v-sum, matrix norms	5
	2D-reductions: m-max/min, row/col-max/min	8
	2D relaxation steps (Jacobi, Gauss–Seidel, ...)	4
	global matrix updates (GJstep, GRstep,..., Mreduce)	6
	1D convolution; intermediate forms of 2D convolution	4
3	matrix multiplication	2
	matrix inversion, Gaussian elimination	2
	intermediate forms of 2D convolution	2
4	2D convolution	1

Table 8.1 A brief summary of the patterns included into the current version of the Basic PARAMAT Pattern Library. No pattern has more than four different templates, most patterns have only one or two. All BLAS routines operating on dense real matrices have been entered.

basic numerical codes from [32] and [30], e.g. Gaussian Elimination (with pivoting), LU decomposition, Simplex, Jacobi relaxation, Gauß–Seidel relaxation, JOR, SOR and CG, iteratively square–rooting a matrix, and others. The results for the Livermore Loops are given in Tab. 8.2. We obtained similar results for the other application codes listed above.

8.4 Pattern Recognition: A Detailed View

Pattern Matching is done in a rather intuitive way. It is supported by a suitable hierarchical representation of the input program and the complexity is diminished by a suitable hierarchical ordering of all patterns. For each template of each pattern there exists a small procedure which tests whether a given piece of program matches this pattern (i.e., an *occurrence* of this pattern), and if yes, it returns an *instance* of this pattern where the formal pattern parameters (*slots*) are bound to the corresponding actual parameters (*slot entries*) occurring in the program piece. [7]

[7] A formal definition of patterns, templates, occurrences, and instances is given in [23].

LL	name	recognized patterns	lp	lr
1	Hydrofragment	GVOP$^{(1)}$	1	1
3	Inner Product	SSP$^{(1)}$	1	1
5	tri-diag. elim., below diag.	FOLRO$^{(1)}$	1	1
8	A.D.I Integration	VHSTAR$^{(1)}$(3x), GVOP$^{(1)}$(3x)	6	6
9	Numerical Integration	GVOP$^{(1)}$	1	1
10	Numerical Differentiation	GVOP$^{(1)}$	19	19
11	First Sum	FOLR$^{(1)}$	1	1
12	First Difference	VHSTAR$^{(1)}$	1	1
13	2D particle in a cell	VCOPY$^{(1)}$(4x),VADD$^{(1)}$(2x),GVOP$^{(1)}$(4x)	17	10
14	1D particle in a cell	GVOP$^{(1)}$(3x),VCOPY$^{(1)}$(1x),VADD$^{(1)}$(2x)	12	6
18	2D explicit hydrodyn. fragm.	MADDMUL$^{(2)}$(2x)	6	2
21	Matrix Product	MMO$^{(3)}$	1	1
22	Planckian Distribution	GVOP$^{(1)}$(2x)	2	2
23	2D implicit hydrodyn. fragm.	MGAUSSSEIDEL$^{(2)}$	1	1
24	1D Minimization	VMINLOC$^{(1)}$	1	1

Table 8.2 Analysis of the Livermore Loops: currently recognizable patterns. lp denotes the number of loops (lp) (after applying loop distribution) occurring in a kernel. lr indicates how many of these can be covered by patterns from the current version of the PARAMAT Library. GVOP$^{(1)}$ denotes a general vector operation, FOLRO$^{(1)}$ and FOLR$^{(1)}$ first order linear recurrences, VHSTAR$^{(1)}$ a 1D difference star; VMINLOC$^{(1)}$ finds in a vector the location with minimal absolute value.

8.4.1 Program Representation

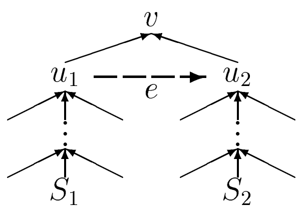
The source program is represented by an attributed syntax tree. The nodes are control statements (`do`, `if`, etc.), assignment statements, or expression operators ($+, -, *, /$ etc.). There is one root node called `main` which represents the highest program control level.

We distinguish between two kinds of directed edges between nodes:

1. *vertical edges:* these are the edges contained in the syntax tree representation of the program, e.g. from all statement nodes corresponding to the body of a `do` statement to the `do` node, from the `then` resp. `else` statement nodes to the corresponding `if` node, from expression tree nodes to their parent nodes and so on.

2. *cross edges:* these edges establish a partial execution order among several child nodes of the same parent node; they are caused by data dependency. If there might

exist a loop–independent[8] data dependence from a statement S_1 to another statement S_2, then a cross edge e must be drawn in the following way: Let v be the lowest common vertical successor of both S_1 and S_2, and let u_1, u_2 be direct vertical predecessors (*'son statements'*) of v such that u_1 is a vertical successor of S_1 (but not of S_2) and u_2 is a vertical successor of S_2 (but not of S_1), see the figure below:



Then the cross edge e must be drawn from u_1 to u_2, implying a partial execution order on the son statements of v. Note that S_1 may be equal to u_1 and S_2 equal to u_2. If $S_1 = S_2$ then there is no edge necessary, of course. If it is not clear at compile time which of several statements is the source of a dependence, then all of them are to be connected with the target node by the way just described. The order among these possible source nodes must be maintained by inserting cross edges following the textual order of the statements in the source program.

Thus the vertical edges form a tree while the cross edges form a directed acyclic graph on each control hierarchy level. For an example, see section 8.4.4.

The cross edges may be extremely useful when trying to recognize patterns from subpatterns which are textually separated by other pieces of code not affecting the relations between these subpatterns. In such "intermixed" computations, statements that belong together (to the same computation thread) are connected by cross edges.

8.4.2 Pattern Hierarchy Graph

The Pattern Hierarchy Graph (PHG) consists of all possible predefined patterns as its nodes. There is a directed edge from one pattern p_1 to another pattern p_2 if p_1 may occur as a subpattern of p_2 (see Fig. 8.1 for an example). Thus the PHG is acyclic.

Each pattern has an *order* number which indicates how many loop nests it contains. Thus an edge from p_1 to p_2 in the PHG implies that $\mathit{order}(p_1) \leq \mathit{order}(p_2)$.

The PHG is called *complete* for a pattern p if it contains p and all possible subpatterns p_i of p and is complete for all p_i.

Usually, a pattern has only a few predecessors and a few successors in the PHG. The pattern matching algorithm only needs to inspect the PHG successor's templates of an already matched pattern p when looking for a possible pattern containing p as a subpattern. That results in a large increase in matching speed compared with simple testing of all predefined templates.

[8]We do not consider loop–carried dependences here because all patterns provided so far allow dependence cycles only from a statement to itself. This is sufficient for most applications if we have applied the restructuring transformations listed in section 8.2 before.

8.4.3 The Matching Algorithm

Starting with *stmtdescend*(*root*), the matching algorithm descends the syntax tree as follows:

---

function *stmtdescend*(*node*)
if *node* is not a leaf **then forall** sons s of *node* (in textual order) **do** *stmtdescend*(s) **od fi**
forall expressions e occurring in *node* **do** *exprdescend*(e) **od**
/⋆ now all vertical predecessors (substatements, subexpressions) of *node* are known ⋆/
/⋆ possible patterns p for *node* are all direct PHG successors (superpatterns)
  of the patterns already computed for the sons s of *node* ⋆/
forall possible patterns p for *node*
do test by *match*(p,*node*) whether there is an instance q of p that matches *node* **od**
replace *node* by q just computed;
reset pointers to and from *node* correctly;
repeat
  forall direct cross predecessors x of *node* in this block
  do /⋆ x has been visited earlier than *node* ⋆/
    test by *merge*(x,*node*) whether there is an instance y
    of a pattern that consists exactly of x and *node*
  od
  replace x and *node* by y just computed;
  reset edges to or from x and *node* to or from y, respectively;
  rename y by *node*
until there are no mergeable predecessors for *node* left.
end *stmtdescend*()

The function *exprdescend*() traverses the expression trees in a similar way.

Each node of the syntax tree is visited exactly once. For each matched node, there is only a constant number of possibilities for choosing a superpattern, and these are tested deterministically one after another, until one of them matches or all fail. Successful matching along a vertical edge or along a cross edge reduces the number of inner nodes in the syntax tree. Thus, the entire algorithm runs in linear time.

This pattern matcher is similar to other bottom–up pattern matching algorithms such as used in automatically generated code generators, e.g. BURG ([14]), TXL ([8]) or OPTRAN ([38]). These systems automatically[9] generate a tree automaton whose state table – if deterministic – corresponds roughly to our PHG. Unfortunately, all automatic systems work only on (syntax) trees; this disables matching along cross edges as required here. Another advantage of our simple pattern matcher is that it can locally deviate from the general scheme to save patterns and matching time, such as we do e.g. for matching difference stars. The high degree of flexibility required for our purpose cannot be supplied by current tree pattern matching kits.

[9] This would make the pattern specification easier if there were many intermediate patterns which only propagate information upwards and never occur in a final matched program. But this does not really occur in our case.

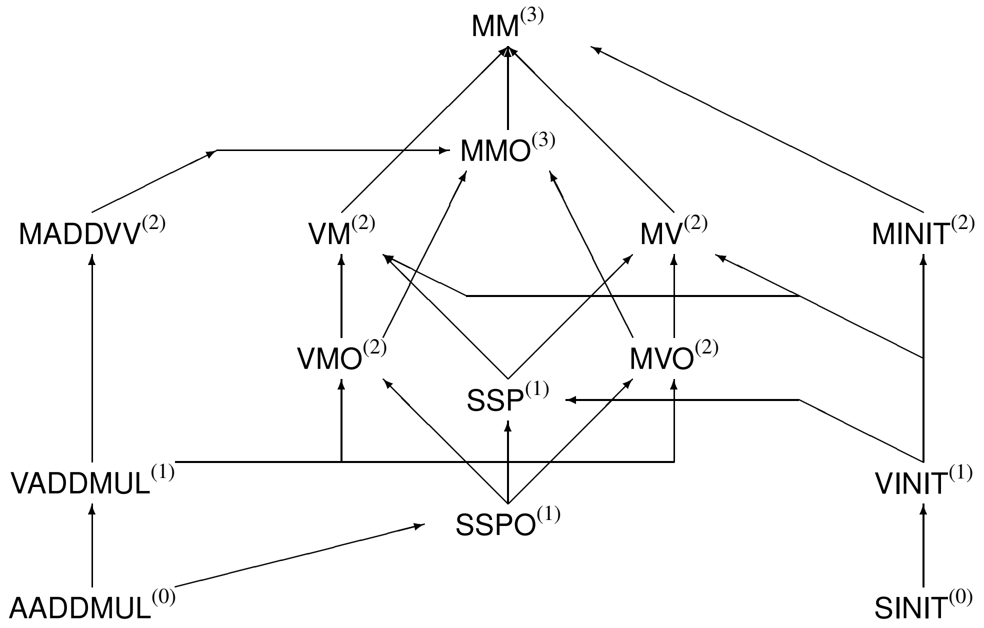


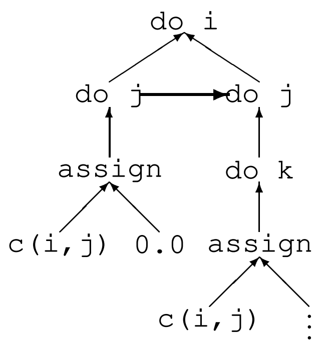
Figure 8.1 The pattern hierarchy graph of Matrix Multiplication. It covers all possible ways how matrix multiplication may be coded without using an auxiliary variable.

8.4.4 Standard Pattern Matching: A simple example

Let us start with a simple example. Matrix multiplication is well suited for this purpose since it is not so trivial but is not made up of too many subpatterns so that its pattern hierarchy graph (Fig. 8.1) remains quite handy.

Suppose the programmer has coded a matrix multiplication in the following way:

```
do i=1,n
  do j=1,m
S1: c(i,j)=0.0
  enddo
  do j=1,m
    do k=1,r
S2:  c(i,j)=c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

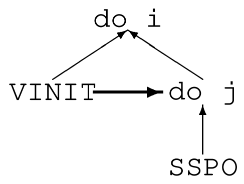


The pattern matcher traverses the syntax tree (see above) by a leftmost depth–first–search. First the assignment node corresponding to S1 will be replaced by the pattern instance `SINIT (c,i,j, 0.0)` since this is the leftmost leaf of the syntax tree (determined by the cross edge caused by the data flow dependence from S1 to S2).

Pursuing this back towards the root, the `do j` loop around S1 will be matched with the SINIT$^{(0)}$ node, resulting in a new pattern instance `VINIT(c(i,*), j,m, 0.0)` (vector initialization).

Then the algorithm descends further towards S2, replaces the assignment statement leaf corresponding to S2 by a pattern instance `AADDMUL(c(i,j),a(i,k),b(k,j))` (accumulatively adding products) and, following the suitable edge in the PHG, merges this with the `do k` loop yielding an SSPO$^{(1)}$ instance (standard scalar product with offset). Now the program corresponding to the modified syntax tree looks as follows:

```
do i=1,n
  VINIT(c(i,*),j,m,0.0)
  do j=1,m
   SSPO(c(i,j),a(i,*),b(*,j),k,r)
  enddo
enddo
```



The `do j` loop can be merged with the SSPO$^{(1)}$ instance resulting in an instance `VMO (c(i,*), a(i,*),b, k,r, j,m)` (vector–matrix multiply with offset). Now the VINIT$^{(1)}$ instance is a mergeable cross predecessor of the VMO$^{(2)}$ instance. Merging them yields a new pattern instance `VM (c(i,*), a(i,*),b, k,r, j,m)` (vector–matrix multiply), which, in turn, can be matched with the `do i` loop resulting in an instance `MM(c,a,b, k,r,j,m,i,n)` (matrix multiply) representing this entire piece of code.

We remark that in this small example, no loop distribution had been applied before. This did not matter since the PHG covers all sensible codings of matrix multiplication, with and without loops being distributed, at the expense of a few more patterns and edges in the PHG. In general, we will have to test whether it is more useful to put more intelligence into the PHG (and thus, perhaps, wasting some space and (compile) time), or to rely on non–failing basic transformations such as loop distribution.

It is clearly no problem to include FORTRAN90's array features and intrinsic array manipulation functions. These just occur as additional templates of their corresponding PARAMAT patterns.

The pattern matcher of PARAMAT is able to detect and undo several former machine-specific optimizations that 'encrypt' the meaning of the code, such as redundant IF conditions, loop blocking, loop unrolling, statement reordering and expression reordering. The next subsections give some examples.

8.4.5 Removing redundant IF statements

This example is taken from the MATMUL routine of the Perfect Club Benchmark program DYFESM (see also [3]) and has already been prepared for pattern matching by the techniques of section 8.2:

```
DO 400 K = 1, N
  DO 100 I = 1, L
    C(I,K) = 0.0
```

```
 100    CONTINUE
        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
 400  CONTINUE
```

The program author has inserted the IF statement during former optimizations. In our case, however, the IF statement - which is obviously redundant regarding semantics - makes the job for pattern recognition unnecessarily harder. But it is no problem to eliminate such a redundant IF statement if we just add a self cycle in the PHG from the $\mathsf{VADDMUL}^{(1)}$ pattern to itself which leaves the $\mathsf{VADDMUL}^{(1)}$ instance unchanged when encountering such an IF condition on one of the operands. After that, we can proceed matching as in the example above.

8.4.6 Loop Rerolling

Since the BLAS routines ([28, 12]) are widely used, it is very important that they are completely recognized by our tool.

Matching has there additionally been made difficult by loop unrolling. Unrolling can appear

- as **replication at the expression level**.

  For an example, consider the following level-1-BLAS routine (after the preprocessing of section 8.2)[10]:

```
      dtemp = 0.0d0
      do 30 i = 1,mod(n,6)
        dtemp = dtemp + dabs(dx(i))
   30 continue
      do 50 i = mod(n,6) + 1, n, 6
        dtemp = dtemp + dabs(dx(i))   + dabs(dx(i+1))
     *              + dabs(dx(i+2)) + dabs(dx(i+3))
     *              + dabs(dx(i+4)) + dabs(dx(i+5))
   50 continue
```

  The original vector sum has been unrolled 5 times for some reason and written as one single expression. To recognize loop 50 as an occurrence of the $\mathsf{VSUM}^{(1)}$ pattern, we must reroll the loop, i.e. undo the former optimization.

[10] At the pattern matching stage, `n` may be known by constant propagation, and so were `mod(n,6)`. If this expression were zero, then the cleanup loop would disappear. For simplicity let us assume here that `n` is either unknown at compile–time (symbolic constant), or that `n` is greater than 6 and `mod(n,6)` is not zero.

Technically, we handle such long right-hand-side expressions by a pattern called MULTIADD$^{(0)}$. MULTIADD$(k, s_1, \ldots, s_k)$ matches a sum of k expressions whose order in summation does not matter[11]. The operands are ordered lexically in the MULTIADD$^{(0)}$ instance. In the example above, this yields that the `dabs(dx(i))` operands appear in ascending order. MULTIADD$^{(0)}$ may refine (specialize) to several other patterns, e.g. here to an accumulating adding sequence AASUM$^{(0)}$. Matching then proceeds via the `i` loop to the corresponding vector instruction VAASUMO$^{(1)}$, and this finally collapses to VSUMO$^{(1)}$ via a refining PHG edge:

```
VSUM(i,1,mod(n,6),1, dtemp,dx(i), abs)
VSUMO(i,mod(n,6)+1,n,1, dtemp,dx(i), abs)
```

Via new PHG edges from VSUMO$^{(1)}$ and VSUM$^{(1)}$ to VSUM$^{(1)}$, these two instances can easily be merged.

- as **replication at the statement level**.

  Consider e.g. the following level-1-BLAS routine (after the preprocessing of section 8.2)[12]:

```
      do 30 i = 1,mod(n,4)
        dy(i) = dy(i) + da*dx(i)
   30 continue
      do 50 i = mod(n,4)+1,n,4
        dy(i)   = dy(i)   + da*dx(i)
        dy(i+1) = dy(i+1) + da*dx(i+1)
        dy(i+2) = dy(i+2) + da*dx(i+2)
        dy(i+3) = dy(i+3) + da*dx(i+3)
   50 continue
```

  The original daxpy loop has been unrolled 3 times for some reason. To recognize this as an occurrence of the VADDMUL$^{(1)}$ pattern, we must reroll the loop. Loop distribution and pattern matching as before generate

```
VADDMUL(1,mod(n,4),1, dy,da,dx)
VADDMUL(mod(n,4)+1,n,4, dy,da,dx)
VADDMUL(mod(n,4)+2,n,4, dy,da,dx)
VADDMUL(mod(n,4)+3,n,4, dy,da,dx)
VADDMUL(mod(n,4)+4,n,4, dy,da,dx)
```

  Now we must first merge the last four VADDMUL$^{(1)}$ instances to get

[11] This is sometimes a problem in FORTRAN since addition or multiplication is in general not commutative, sometimes even not associative for FORTRAN reals. We however remind that (1) this is only for easier pattern recognition, and the original operand order may be rearranged at code generation time, and (2) this problem is also ignored for many parallelizing transformations such as summing up the `VSUM` operands in a tree-like fashion for parallel execution.

[12] For simplicity let us assume again that `n` is symbolic or greater than 4 and `mod(n,4)` is not zero.

```
VADDMUL(1,mod(n,4),1, dy,da,dx)
VADDMUL(mod(n,4)+1,n,1, dy,da,dx)
```

this is technically arranged by a self-cycle in the PHG from VADDMUL$^{(1)}$ to itself[13]. After that, it is no problem to merge these two instances into a single VADDMUL$^{(1)}$ instance via another PHG-self-cycle for VADDMUL$^{(1)}$.

- as **blocking at the loop level** (also known as strip mining (one-dimensional) or tiling (more-dimensional)).

  This often occurs in old codes which have been tuned for efficient use of caches or vector registers (see e.g. [22]). For daxpy, this looks as follows:

```
   if( mod(n,k) .ne. 0 ) then
   do 30 i = 1,mod(n,k)
     dy(i) = dy(i) + da*dx(i)
30 continue
   do 50 i = mod(n,k)+1,n,k
    do 40 j=i,i+k-1
     dy(j) = dy(j) + da*dx(j)
40  continue
50 continue
```

  Here matching of the second loop nest is also enabled via a PHG self-cycle of the VADDMUL$^{(1)}$ pattern[14]:

```
   VADDMUL(i,1,mod(n,k),1, dy,da,dx)
   do 50 i = mod(n,k)+1,n,k
    VADDMUL(j,i,i+k-1,1, dy,da,dx)
50 continue
```

  Via this PHG edge, the second VADDMUL$^{(1)}$ instance merges with the `i` loop into

```
VADDMUL(i,1,mod(n,k),1, dy,da,dx)
VADDMUL(i,mod(n,k),n,1, dy,da,dx)
```

  which then collapses into a single VADDMUL$^{(1)}$ as in the previous example.

Using these techniques, we can completely match the Level 1 BLAS routines:

[13] More exactly: there is an auxiliary ‘collecting’ pattern for VADDMUL$^{(1)}$ (and, correspondingly, also for other patterns such as VADD$^{(1)}$, VSUM$^{(1)}$, ...) which just tries to merge ‘similar’ VADDMUL$^{(1)}$ instances and which is also robust against changes in the order of the sequence of VADDMUL$^{(1)}$ instances within the program representation. It is also possible to have the cleanup loop at the end of the sequence, as in [22].

[14] For simplicity, let us assume again that the value of `mod(n,k)` is unknown at compile time or not zero.

BLAS1 routine	pattern covering this routine
dasum	VSUM$^{(1)}$(with abs flag set)
daxpy	VADDMUL$^{(1)}$
dcopy	VCOPY$^{(1)}$
ddot	VSUM$^{(1)}$
dnrm2	ENORM$^{(1)}$
drot	VROT$^{(1)}$
drotg	(scalar computation)
dscal	SV$^{(1)}$
dswap	VSWAP$^{(1)}$
idamax	VMAXLOC$^{(1)}$(with abs flag set)

8.4.7 Difference Stars

Difference stars appear e.g. at PDE discretization (in the context of grid relaxations (Jacobi, Gauß–Seidel), as coarser-to-finer grid interpolation and so on).

Example: (Jacobi relaxation)

```
   do 10 j=2,n-1
    do 10 i=2,n-1
     uhelp(i,j) = (1-omega)*u(i,j) + omega*0.25*( f(i,j)
  *        + u(i-1,j) + u(i+1,j) + u(i,j+1) + u(i,j-1) )
10 continue
```

PARAMAT provides difference star patterns for one (HSTAR$^{(0)}$) and two (STAR$^{(0)}$) dimensions on the expression level. These are both specializations of the MULTIADD$^{(0)}$ pattern mentionned above. The most important *relative grid positions* of a 1D or 2D difference star are numbered from [i-1,j-1] (= pos. 1) to [i+1,j+1] (=pos. 9) in lexicographic order:

```
[i-1]       [i]       [i+1]          ---> HSTAR  4  5  6

[i-1,j-1]   [i-1,j]   [i-1,j+1]                  1  2  3
[i,j-1]     [i,j]     [i,j+1]        ---> STAR   4  5  6
[i+1,j-1]   [i+1,j]   [i+1,j+1]                  7  8  9
```

The i's and j's in these index patterns may be preceded by a constant integer scaling factor (restriction factor); in the example above, this factor is 1, but for coarser-to-finer (or vice versa) grid interpolations occurring e.g. in multigrid programs, a factor of 2 can occur.

The slots of both patterns are:

- C[1],...,C[9]: The immediate coefficient expressions. They may also contain arrays indexed by [i,j]. They are zero if the corresponding star position is not present.
- Cges : common factor for all star positions
- C5 : coefficient of an extra occurrence of the rhs array
- LHS : lhs array name, may be equal to RHS

- starvar : rhs array indexed by this star
- F: additional quasiscalar array ($\neq starvar$) access indexed in [i,j]
- CF: factor for F; 1.0 if not present
- RI, RJ: restriction factors (must occur in all rhs accesses)
- dim1, ivar1, dim2, ivar2 : dimensions and names of i and j.

The right–hand–side expression may be permuted in many variants, but in case it should match a $\mathsf{HSTAR}^{(0)}$ or $\mathsf{STAR}^{(0)}$, there always must be an occurrence of $\mathsf{MULTIADD}^{(0)}$. A $\mathsf{MULTIADD}^{(0)}$ instance which meets the condition that among its operands there is an array occurring at least 2 times, with different indexing as permitted above, will be refined to $\mathsf{HSTAR}^{(0)}$or $\mathsf{STAR}^{(0)}$instance by calling an auxiliary routine called *stargazer*. This routine successively tries to fill in more slots while ascending the rhs expression tree; it cycles to itself (while consuming text) as long as it can enter a new operand into the slots, until it reaches the assignment[15].

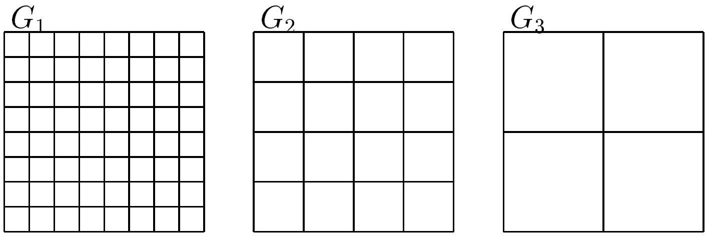
For the example above (Jacobi), the slots are filled as follows:

- C[1]=0.0, C[2]=1.0, C[3]=0.0, C[4]=1.0, C[5]= 0.0, . . ., C[8]=0.0, C[9]=1.0
- Cges=`0.25*omega`, C5=`1-omega`,
- LHS = `uhelp`, starvar = `u`, F = `f`, CF = 1.0
- RI=1, RJ=1, dim1=1, ivar1=`i`, dim2=2, ivar2=`j`

If successful, *stargazer* returns either a $\mathsf{STAR}^{(0)}$ (full star) or a $\mathsf{HSTAR}^{(0)}$ (half star, i.e. there are only either positions 2,5,8 or 4,5,6 filled by operands). It is now no problem any more to detect Jacobi, Gauß–Seidel or other relaxation steps when going on matching the indexing loops around the star.

8.4.8 Beyond standard matching: Identification of multigrid hierarchies

Multigrid programs (see e.g. [35]) operate on a grid hierarchy of several grids of different size. For instance, in the figure below, G_1 is the finest grid (64 elements) and G_3 the coarsest one (4 elements):



The main components of multigrid programs are:

- a relaxation algorithm,
- a finer-to-coarser-grid restriction operator,
- a coarse grid solution algorithm, and

[15] This procedure deviates locally from the standard pattern matching method we described above. It arises that standard pattern matching can recognize complex stars only with large space and time overhead. By this tricky procedure, we make star recognition handy. One may regard *stargazer* as an ‘intelligent pattern’ within the PHG.

- a coarser-to-finer-grid interpolation operation.

Continuing the example above, consider the following restriction operation (after preprocessing):

```
do i=1,3
 do j=1,3
  G2(i,j)= f1(2*i,2*j) - 4.0*G1(2*i,2*j) + G1(2*i,2*j-1)
         + G1(2*i,2*j+1) + G1(2*i-1,2*j) + G1(2*i+1,2*j)
```

This code will be recognized by *stargazer*, as discussed above, as a full $\mathsf{STAR}^{(0)}$with restriction factors RI=RJ=2 and finally matches with the surrounding indexing loops to a $\mathsf{RESTR}^{(2)}$ instance. This also establishes further knowledge on the grid hierarchy. This additional knowledge may be used to find out the grid hierarchy. Our current research investigates this direction.

Another difficulty may arise here due to former (space) optimizations of the underlying code. Real multigrid programs are often coded by storing all grid hierarchy levels of the same array into a large linear workarray:

G_1	G_2	G_3	$\cdots$

Due to procedure inlining in the preprocessing phase, the code submitted to the pattern matcher contains only one-dimensional grid accesses and looks quite awful. For this case, PARAMAT must provide a second set of templates for all grid relaxation patterns which handle the linearized versions[16]. We are currently working on this issue. We are confident that PARAMAT will finally be able to detect the whole underlying work space concept of a multigrid code even in this case. Here, in particular, knowledge about the hidden grid hierarchy is essential for the computation of efficient array distributions (see also [16], there user directives tell the compiler (SUPERB) about the grid hierarchy within the working array) since — if not detected — the linear array will most probably be distributed over the processors as equally sized slices, and that results in suboptimal load balancing and unnecessary communication.

8.5 A Parallel Algorithm for each Pattern

For each predefined pattern there exists a suitable parallel algorithm implementing this pattern, parameterized by the problem size (e.g. matrix extents) and, if necessary, also by the data partitionings for the arrays involved in this pattern.

A major benefit of this technique is that once the pattern is recognized, the best possible algorithm for this pattern will be chosen. For instance, if the programmer has implemented a Gauß–Seidel relaxation in a wave–front manner, the system should choose

[16]If the workspace array is accessed indirectly via index vectors, pattern matching is disabled as dependence analysis is unable to cope with index vectors unless constant (array) propagation supplies the values of the index vector elements at compile time.

another implementation which is better suited for efficient parallelization, e.g. a red–black scheme (which has the same convergence property as the wavefront scheme) or replace it by the double number of Jacobi– or Gauß–Seidel–Jacobi hybrid iterations[17].

Another occasion for algorithm replacement are simple linear recurrences. Consider for instance the following piece of code:

```
        X[1] = A[1]
        DO 1 I=2,1000
           X[I] = A[I] + B[I] * X[I-1]
1       CONTINUE
```

Independent of the data partitioning, this code is doomed to sequential execution due to the loop–carried data dependence on array X. Once recognized as an instance of the $\mathsf{FOLR}^{(1)}$ pattern (First Order Linear Recurrence), this piece of code will be replaced by recursive doubling techniques[18] described in [26].

For some patterns, the programming environment may supply highly–optimized parallel implementations, e.g. for $\mathsf{VSUM}^{(1)}$ (global summation of vector elements). In such a case the pattern instance is simply converted into a runtime library call.

If the target machine has, for example, pipelined vector units as e.g. the TMC CM5, then there should be done code optimization for vector register allocation. This is particularly worth doing for vector operations, i.e. for instances of the $\mathsf{VADD}^{(1)}$, $\mathsf{VMUL}^{(1)}$, ..., $\mathsf{GVOP}^{(1)}$ patterns. Especially for machines with small register files, the vector register allocation and scheduling techniques from [22] can speed up execution of basic blocks of vector instructions considerably.

Thus, algorithm replacement enables sophisticated target–machine specific code optimizations that are hidden for the non-expert user. As remarked above, it is no problem to re-introduce here optimizations that have been removed before or within the pattern recognition phase. It is also possible to locally deviate here from the strict owner-computes-rule by selecting suitable implementations.

For each parallel implementation of a pattern (located in a large implementation library) there is an assigned cost function which is also parameterized by problem size and array partitionings. This function models the run time behaviour of the algorithm under consideration. Section 8.7 explains how it is determined.

8.6 Alignment and Partitioning

The problem of array partitioning consists of two steps. First, it must be determined how arrays should be *aligned* with each other, i.e. which elements of different arrays should be mapped together to the same (virtual) processor (if possible) to minimize interprocessor communication. The alignment preferences are induced by the array references

[17]The latter replacements should be a priori allowed by the user. We expect that the average user does not want to compare Gauß–Seidel with Jacobi but to get the actually fastest parallel implementation — independent of a particular relaxation coding.

[18]The optimal number of recursive doubling steps depends on the number of iterations and on the message startup time of the target machine. For small problem sizes, sequential execution will be faster.

pattern	algorithm	alignment recomm.	distribution recomm.
$\mathsf{MCOPY}^{(2)}(A,B)$	matrix copy	$A \equiv B$	arbitrarily
$\mathsf{VCOPY}^{(1)}(V,W)$	vector copy	$V \equiv W$	arbitrarily
$\mathsf{MJACOBI}^{(2)}(A,B)$	one Jacobian relax. step	$A \equiv B$	quadratic blocks
$\mathsf{MM}^{(3)}(C,A,B)$	matrix multiplication	$A \equiv C \ \vee \ B \equiv C$	A by row, B by col
$\mathsf{MTRANSP}^{(2)}(A,B)$	matrix transpose	$A \equiv^{-1} B$	arbitrarily
$\mathsf{VSUM}^{(1)}(s,V)$	vector summation	arbitrarily	arbitrarily
$\mathsf{SSP}^{(1)}(s,V,W)$	standard scalar product	$V \equiv W$	arbitrarily

Table 8.3 Array alignment and distribution recommendations for some patterns

occurring in the source program. Second, the array elements must be *distributed*, i.e. actually mapped to a concrete processing element of the (physical) target machine. At this stage, aligned array sections can be handled as an entity.

For alignment and partitioning issues, we will use well–known techniques introduced by Li and Chen ([29]), Knobe et al. ([24, 25]) or Wholey ([37]). The cost estimate functions in these approaches will be replaced by our own cost functions being described in the next section.

The problem of determining optimal data alignment and distribution is well–known to be NP–complete (cf. [29]), thus automatic partitioning may take exponential time in the worst case. That is why we intend to limit the number of distribution alternatives severely[19]. Furthermore, we limit the number of partitionable array dimensions to 2. Together with the simplification of the source program by pattern matching, these restrictions enable the application of a branch–and–bound search for the optimal distribution, as done in [19].

For each pattern there generally exists one locally optimal data distribution scheme. E.g. for matrix multiplication $C = A \cdot B$, matrix A should be distributed by row and matrix B should be distributed by column, and furthermore, C should be aligned with either A or B. PARAMAT provides a default recommendation for alignment and one for array distribution for each pattern[20]. If these recommendations cause an alignment or distribution conflict throughout the program (and they will usually do), the given recommendations will be the first choices when searching for the optimal data distribution.

Some recommendation examples are summarized in Tab. 8.3.

The points between the pattern instances in the final matched program representation are natural places for possible redistribution of arrays. In [21] an interesting method for (static) redistribution has been presented (see also [7] in this volume). We will include it

[19]In fact we will only allow the following six possibilities (for two-dimensional arrays): row-/column-/block-contiguous and row- or column–cyclic (cf. e.g. [17]). For all the applications considered so far these are enough to supply good distributions.

[20]Note that the alignment recommendation depends only on the pattern itself whereas the distribution recommendation is, in general, also dependent on the target machine and on the problem size (cf. [21]).

into the PARAMAT system with small modifications. The details will be given in a later paper.

8.7 Determining Cost Functions: Estimating and Benchmarking

A simple way to determine the run time of a given parallel algorithm parameterized by problem size and array partitions is to estimate it from the basic computation and communication statements occurring in that algorithm (see e.g. [13]).

In reality, however, it seems that these estimations very rarely meet the actual run time since a parallel computer such as the INTEL iPSC/860 behaves as a chaotic system, its performance heavily depending on the actual network traffic, which, in turn, depends on the algorithm under consideration. This problem has been adressed in [2]. There performance estimation is based on benchmarking of simple communication patterns and propagating this information 'up' through the program's syntax tree.

PARAMAT will be able to inspect precomputed tables of *really measured* run times for *all* pattern implementations (especially, for high–level patterns!), as indicated above. We expect the accuracy of performance prediction being substantially improved by this method.

Problems with performance prediction 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 or must be reloaded first. This scenario is influenced by all former operations. Here more research has to be done.

8.8 Implementation and Future Extensions

The overall PARAMAT system is outlined in Fig. 8.2. Note that the time–consuming work concerning parallel algorithm development (either optimized SPMD routines or just operating system calls) and benchmarking their run times for all possible array distributions has not to be performed at compile time but at an earlier stage (at compiler generation time). We intend to develop an automatic benchmarking tool that does this tedious job for us.

Furthermore it will be useful to arrange the Pattern Library as a hierarchical collection of modules which may be individually composed for special application areas.

The pattern recognition tool — as described in this paper — has been implemented and tested. The current (July 1993) implementation consists of around 10000 lines of C code and reliably recognizes 68 nontrivial patterns with 117 nontrivial templates[21], including special routines for loop rerolling and identification of difference stars. The high degree of robustness against loop interchange, loop distribution, loop unrolling and statement reordering has been exemplified in practice.

[21] Each template is technically represented by 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.

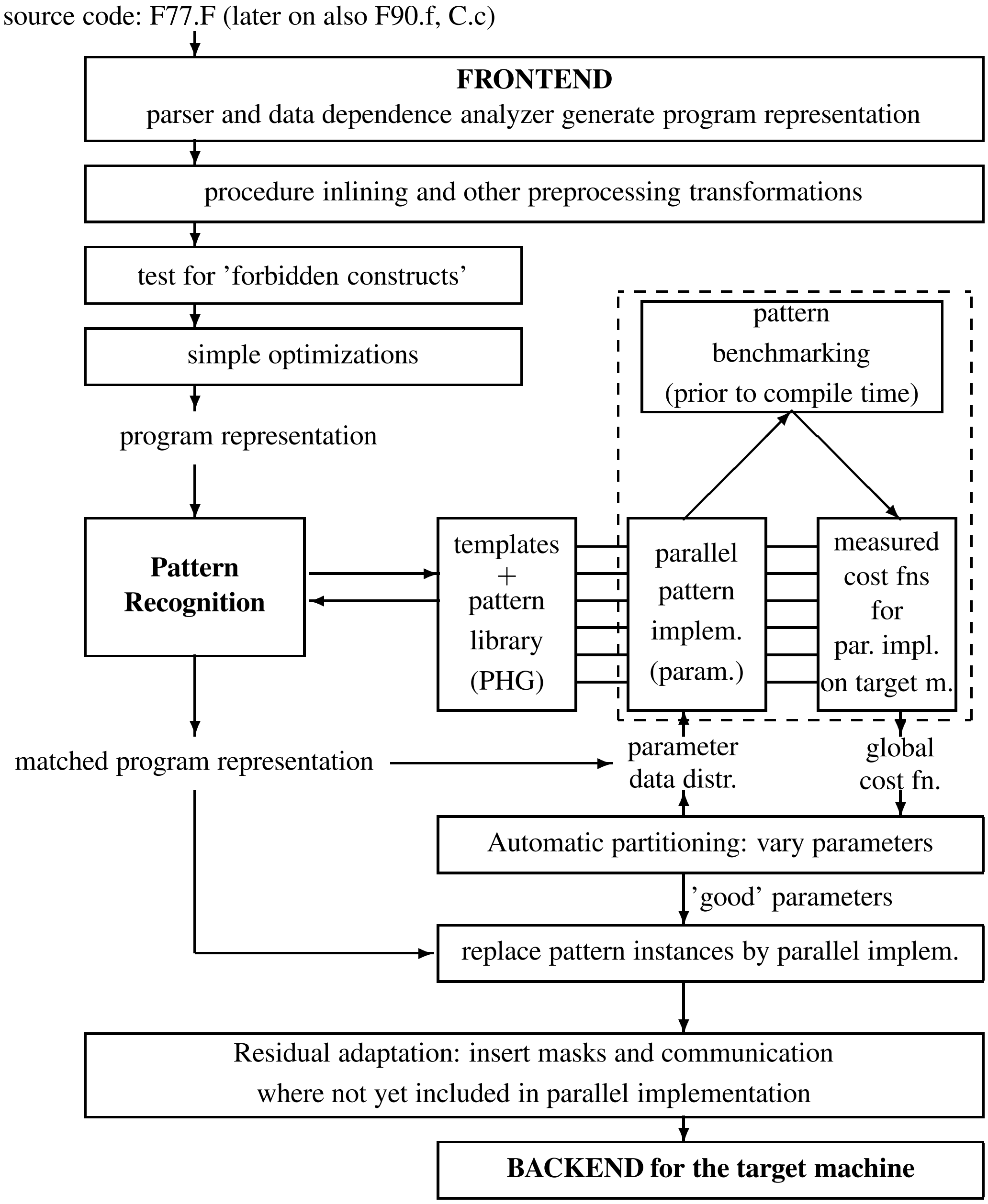


Figure 8.2 Block diagram of the overall PARAMAT system. Only the components enclosed in the dashed rectangle (and the back end, of course) are dependent on the target machine. Parallel algorithms for each pattern and their run–time approximations, parameterized by the data partitionings of the arrays involved in the resp. pattern instance, have to be precomputed once for each hardware constellation (pattern benchmarking) *before* compile time.

In order to determine cross edges properly, we have implemented a simple version of exact array data flow analysis. This (in general very problematic) analysis is here highly simplified by the exact data flow information[22] supplied with the instances of recognized patterns.

More patterns can easily be added. We plan to implement patterns from image processing algorithms in the near future. We will also try to integrate loop distribution into the pattern recognition tool.

8.9 Conclusions

We have outlined the PARAMAT system which performs automatic parallelization of a restricted but quite important class of numerical programs. Parallelization is done by pattern recognition and local algorithm replacement. The pattern recognition algorithm with the hierarchical pattern base described above is robust against common semantics preserving code modifications and even decrypts many former optimization transformations of dusty deck sources. Pattern recognition simplifies the program graph and thus alleviates global optimizations such as determining optimal array alignment and partitioning. Pattern recognition enables optimal use of the highly optimized operating system routines supplied with the target hardware environment. It also supports better performance prediction accuracy by benchmarking higher–level patterns. Beyond automatic parallelization, pattern recognition may be of great use at reengineering sequential dusty–deck codes for porting them to other languages or target machines.

Algorithm replacement also means automatically performing optimizing transformations of the code. The combination of pattern recognition and algorithm replacement makes the experience of parallel programming and optimization experts accessible to all scientific Fortran 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 has not to recognize his 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 code modifications and hides the parallelization details from the user.

The PARAMAT system is limited in some ways. First, it is unable to process *all* source programs. This is taken into account to obtain a really automatic parallelizing compiler *without any* interaction or directives supplied by the user at compile time. Second, the search space for possible array distributions has to be severely restricted to maintain acceptable compile time. In general, heuristics will have to be applied.

A prototype of PARAMAT is currently being implemented at Saarbrücken university. The implementation of the pattern recognition tool has reached the state described in this paper. The first target machine will be the Intel iPSC/860.

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 run time approximation functions can be automated by a universal benchmarking tool.

[22] For each (array) slot, there is an associated *descriptor* that exactly (symbolically) describes which elements are accessed in the array.

Changing the hardware platform means only to load another base of parallel pattern implementations and their run time functions. Thus the PARAMAT system will always be up to date for the latest hardware environments available.

Acknowledgements

We would like to thank Prof. A. K. Louis, Prof. W. J. Paul, Prof. R. Wilhelm, Prof. H. Zima, Barbara Chapman, Christian Ferdinand and all AP'93 participants for their comments and suggestions to this work.

References

[1] David H. Bailey and John T. Barton. The NAS Kernel Benchmark Program. Numerical Aerodynamic Simulations Systems Division, NASA Ames Research Center, June 1986.

[2] 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.

[3] Michael Berry (Editor). Scientific Workload Characterization by Loop Based Analyses. *Performance Evaluation Review*, 19, February 1992.

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

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

[6] 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.

[7] Alan Carle, Ken Kennedy, Ulrich Kremer, and John Mellor–Crummey. Automatic Data Layout for Distributed Memory Machines in the D Programming Environment. In *Proc. of AP'93 Int. Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction, Saarbrücken, Germany*, March 1993.

[8] Ian A. Carmichael and James R. Cordy. *TXL - Tree Transformational Language Syntax and Informal Semantics*. Dept. of Computing and Information Science, Queen's University at Kingston, Canada, February 1992.

[9] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. In *Third Workshop on Compilers for Parallel Computers*, 1992.

[10] Barbara M. Chapman, Heinz Herbeck, and Hans P. Zima. Automatic Support for Data Distribution. Technical Report ACPC/TR 91-14, Austrian Center for Parallel Computation, July 1991.

[11] Barbara M. Chapman, Heinz Herbeck, and Hans P. Zima. Knowledge-based Parallelization for Distributed Memory Systems. Technical Report ACPC/TR 91-11, Austrian Center for Parallel Computation, April 1991.

[12] 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.

[13] Thomas Fahringer, Roman Blasko, and Hans Zima. Automatic Performance Prediction to Support Parallelization of Fortran Programs for Massively Parallel Systems. In *Int. ACM Conference on Supercomputing*, 1992. Washington DC.

[14] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

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

[16] Michael Gerndt. Parallelization of Multigrid Programs in SUPERB. Technical Report ACPC/TR 90-6, Austrian Center for Parallel Computation, October 1990.

[17] Manish Gupta and Prithviraj Banerjee. Automatic Data Partitioning on Distributed Memory Multiprocessors. Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Oct. 1990.

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

[19] Roman Hayer. Automatische Parallelisierung, Teil 2: Automatische Datenaufteilung für Parallelrechner mit verteiltem Speicher. Master thesis, Universität Saarbrücken, 1993.

[20] 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.

[21] Ken Kennedy and Ulrich Kremer. Automatic Data Alignment and Distribution for Loosely Synchronous Problems in an Interactive Programming Environment. Technical Report COMP TR91-155, Rice University, April 1991. See also this volume.

[22] C.W. Keßler, W.J. Paul, and T. Rauber. Scheduling Vector Straight Line Code on Vector Processors. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, Springer Workshops in Computing Series, pages 77–91, 1992.

[23] Christoph W. Keßler. The Basic PARAMAT Pattern Library, May 1993.

[24] 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.

[25] 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.

[26] 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), August 1973.

[27] Wojtek Kozaczynski, Jim Ning, and Tom Sarver. Program concept recognition. In *Proc. of KBSE'92 Seventh Knowledge-Based Software Engineering Conference*, pages 216–225, 1992.

[28] 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.

[29] 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.

[30] A. K. Louis. Parallele Numerik. Course script and selected programs, unpublished, Universität Saarbrücken, 1992.

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

[32] Silvia M. Müller. Die Auswirkung der Startup-Zeit auf die Leistung paralleler Rechner bei numerischen Anwendungen. Master thesis, Universität Saarbrücken, 1989.

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

[34] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Runtime Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[35] Klaus Stüben and Ulrich Trottenberg. Multigrid methods: Fundamental algorithms, model problem analysis and applications. In *Springer Lecture Notes in Mathematics, Vol. 960*, 1982.

[36] Ko-Yang Wang and Dennis Gannon. Applying AI Techniques to Program Optimization for Parallel Computers. In Kai Hwang and D. DeGroot, editors, *Parallel Processing for Supercomputers and Artificial Intelligence*, pages 441–485, 1989.

[37] Skef Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

[38] Reinhard Wilhelm. Computation and Use of Data Flow Information in Optimizing Compilers. *Acta Informatica*, 12:209–225, 1979.

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