# Symbolic Array Data Flow Analysis and Pattern Recognition in Numerical Codes

Christoph W. Keßler*

**Abstract**

The PARAMAT approach to fully automatic distributed–memory 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 give a fast and powerful pattern recognition algorithm which tries to locally recover the semantics of the program while being robust against many common code modifications like loop distribution, loop interchange, loop blocking or loop unrolling. Third, we use the restored program semantics information to guide data partitioning, run time prediction and sophisticated optimizing code transformations including algorithm replacement.

In this paper we focus on the symbolic array dataflow analysis techniques of PARAMAT's pattern recognition tool. We introduce a compact symbolic access descriptor and use it to compute array data flow that guides the recognition process. As an example, we show the application of this method to recognize and reroll unrolled loops.

## Introduction

Distributed memory systems (DMS) are well–known to be difficult to program. The most important reasons are: (1) Parallel code must contain explicit message passing statements, and programming explicit message passing is complex, tedious and error-prone. (2) The efficiency of the target program heavily depends on choosing a suitable data distribution (and sometimes, also redistribution) — for larger applications, this is a very hard problem. (3) SPMD programs must in general be transformed to be efficient — just applying the owner-computes-rule will usually not suffice. There is, however, no guidance in which optimizing transformations to choose, and in which order to apply them. (4) Run time prediction for non-trivial codes on real machines is a very problematic matter for reasons of network contention, message protocols, buffering, undocumented hardware features, and others.

Message passing statements can be generated automatically today by *semi-automatic* parallelization. The user has to provide data distribution and optimizing

*Graduiertenkolleg Informatik, Universität des Saarlandes, P.O. Box 151150, D-66041 Saarbrücken, Germany. e-mail: `kessler@cs.uni-sb.de`

transformations, either in the form of interactive commands in tools like SUPERB [Ger89] or in the form of language constructs or compiler directives in an explicitly parallel language such as Fortran-D [HKT91], Vienna Fortran [CMZ92], HPF [HPF93] and others.

Nevertheless, there are hard problems involved in automatic data distribution and redistribution, in automatic guidance in optimizing transformations, and in suitably accurate performance prediction. These problems arise from the fact that there is often not sufficient knowledge available on the source program and on the target machine characteristics. So what we should do to make fully automatic DMS parallelization feasible at all is to gain as much of this knowledge as possible. This works, of course, not for all programs.

Many numerical programs are, however, in particular good-natured for this purpose. We have examined a lot of typical application codes that are to be run on distributed memory systems. As a result, we have observed that there is only a rather limited number of typical programming idioms, called *patterns*, that often occur in these programs, in particular in the time–consuming inner loops, and we have collected around 150 typical patterns and also typical modifications (transformations) of them, called *templates*, in a *basic pattern library* [Keß93b]. We claim that these few patterns cover large parts — especially the time–critical ones — of numerical application codes that are actually run on distributed–memory multiprocessors. We exemplified this by examining the source codes of standard numerical benchmark programs [KP93, Keß93b].

From this observation, we construct a sophisticated automatic DMS parallelizer called PARAMAT [KP93, Keß93a, Keß93b] with the following key ideas: A pattern recognition tool working fast and reliably locally recognizes pattern incarnations and replaces them by a pattern *instance*, looking similar to a Fortran90 intrinsic function call. By this information, we can infer a lot of additional knowledge on mathematical properties, efficient implementations, favorable data distributions and run time behaviour, to guide a sophisticated parallelization process including high–level program transformations up to local algorithm replacement.

In this paper we focus on the symbolic array dataflow analysis techniques of PARAMAT's pattern recognition tool that are responsible for a significant part of its performance. We introduce a compact symbolic access descriptor. We use it for two purposes: to analyze data dependence and to compute array data flow that guides the recognition process. We give an example for the application of this method by showing how to recognize unrolled loops and how to reroll them to obtain a target-machine-independent program representation.

# Pattern Recognition

In order to be able to do a proper job, the source program must be preprocessed first to make the program as *explicit* as possible, for instance to propagate constant expressions forward to the place where they are needed, to get rid of induction temporaries, to inline procedures, to make the program well-structured by getting rid of `GOTO`'s, to simplify arrays with dimensions of low extent, etc.

**Pattern Hierarchy Graph** The pattern library is hierarchically organized. We represent it as a directed acyclic graph, the pattern hierarchy graph (PHG). A PHG edge connects a pattern $p_1$ to a pattern $p_2$ iff $p_1$ may occur as a direct subpattern in a template of $p_2$. The PHG edges are used to trigger pattern matching. 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 faster pattern recognition compared with simple testing of all predefined templates.

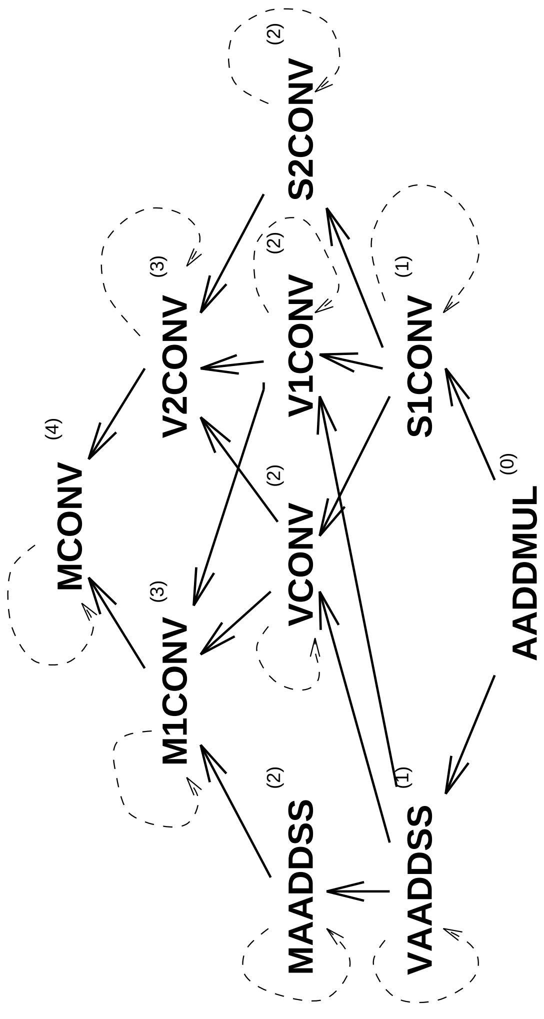
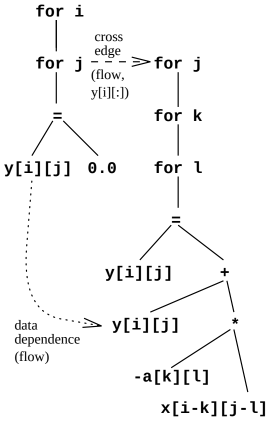



Fig. 1: The PHG of 1D ($\mathsf{VCONV}^{(2)}$) and 2D ($\mathsf{MCONV}^{(4)}$) convolution. Each solid edge into a pattern stands for a particular 'vertical' template of this pattern; the dashed edges represent templates for matching along cross edges. For the reason of improved performance, self-cycles from a pattern to itself are admitted.

**Pattern Recognition Strategy** The source program's syntax tree is traversed bottom-up and from the left to the right. A recognized node is replaced by a pattern instance. Each node is visited only once, and for each node, there are only a few possible patterns due to the PHG structure. Thus, matching time is guaranteed to be linear in the size of the source program.

We have designed the templates in such a way that pattern recognition is robust against typical code transformations like expression reordering, statement reordering, loop interchange, loop distribution, loop unrolling, loop peeling, loop blocking or introduction of temporary variables.

**Example** The PHG of 2D convolution is given in Fig. 1. Suppose the programmer has coded a 2D convolution as follows:

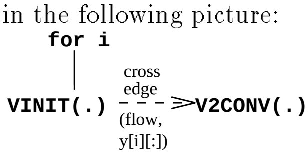



```
  for (i = 1; i <= u1-1; i++) {
    for (j = 1; j <= u2-1; j++)
S1:   y[i][j] = 0.0;
    for (j = 1; j <= u2-1; j++)
      for (k = 1; k <= u3-1; k++)
        for (l = 1; l <= u4-1; l++)
S2:       y[i][j]=y[i][j]-a[k][l]*x[i-k][j-l];}
```

The pattern matcher traverses the syntax tree in leftmost postorder. First the assignment node at S1 will be replaced by the pattern instance `SINIT(y[i][j],0.0)`. Thereafter, the `for j` loop around S1 will be matched as a new pattern instance `VINIT(j, c(i,:), 0.0)` (vector init.). The *range* slot for the loop variable `j` also contains information on loop bounds and step size.

Then the algorithm descends towards S2, replaces the assignment statement at S2 by `AADDMUL(y[i][j], -a[k][l], x[i-k][j-l])` and, following the suitable edge in the PHG, merges it with the `do l` loop giving the instance `S1CONV(l, y[i][j], y[i][j], -a[k][:], x[i-k][j-:])`. Note that `a[k][:]` denotes a vector, as well as `x[i-k][j-:]`. ':' stands for a ranging loop variable. From its range, the spanned vector elements can easily be computed. The `S1CONV` instance matches then with the `for k` loop into `S2CONV(l,k, y[i][j], y[i][j], -a[:][:], x[i-:][j-:])`, with `a` and `x` now being matrices. This instance, in turn, can be matched with the `for j` loop, as shown in the following picture:

```
  for (i=1; i<=u1-1; i++) {
S1':  VINIT(j, y[i][:], 0.0);
S2':  V2CONV(j, k, l, y[i][:], y[i][:],
             -a[:][:], x[i-:][:-:]);  }
```




There is a cross edge from S1' to S2' of type FLOW, caused by the vector `y[i][:]` (`i` is a constant at this point; the second dimension of `y` is bound by `j` ranging from `1` to `u2-1`). For V2CONV$^{(3)}$, VINIT$^{(1)}$ is a mergeable cross predecessor, according to the matching rules supplied with the PHG. Merging these two instances along the cross edge yields a new pattern instance `V2CONV(j, k, l, y[i][:], 0.0, -a[:][:], x[i-:][:-:])`. Matching this with the `do i` loop gives `MCONV(i, j, k, l, y[:][:], 0.0, -a[:][:], x[:-:][:-:])` representing the entire piece of

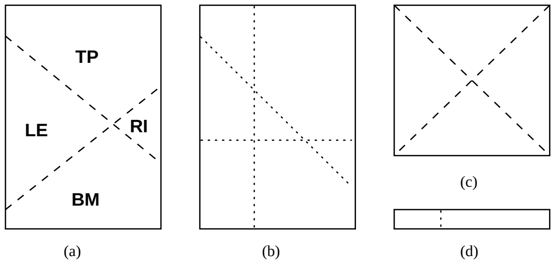



Fig. 2: Possible descriptor variations: (a) the four areas for tridiagonal matrices, (b) excluded row, column or diagonal for rectangular matrices, (c) diagonal or counterdiagonal vector in a matrix, and (d) excluded position in a vector.

code. In the PHG (Fig. 1), the pattern matching algorithm has taken the path $\mathsf{AADDMUL}^{(0)}$, $\mathsf{S1CONV}^{(1)}$, $\mathsf{S2CONV}^{(2)}$, $\mathsf{V2CONV}^{(3)}$, $\mathsf{MCONV}^{(4)}$ which corresponds to this particular coding of 2D convolution.

There are many ways to code 2D convolution: The four loops may be permuted in any order. Each coding then results in a different path being taken by the pattern matcher in the PHG towards $\mathsf{MCONV}^{(4)}$. Moreover, in each of these cases, the initialization may be more or less distributed over the `for i` and the `for j` loop. Furthermore, each loop may be unrolled or blocked, and additional conditions may be inserted to avoid redundant computations. All these coding possibilities (exponentially many in terms of PHG size) are covered by the PHG of Fig. 1: Only 9 patterns with 25 templates (16 vertical, 9 cross) altogether suffice to represent all codings.

# A compact descriptor for trapezoidal matrices

For each slot of a pattern instance that contains an array reference, we compute a compact *descriptor* that exactly summarizes which part of the array's index domain is accessed, e.g., trapezoidal or triangular parts of a matrix, diagonal vectors, or a rectangular region, perhaps with excluded row, column or diagonal vectors. Although the descriptor is, in principle, not limited to matrices, this two–dimensional case is most interesting for our application domain.

**Descriptor definition** Let us consider a subtree of the syntax tree representation of the sequential source program. An integer variable is called *ranging* if it is the loop variable[1] of a loop occuring in this subtree. For an occurrence of a (possibly higher–dimensional) array $A$, call a dimension $d$ of $A$ ranging if there occurs at least one ranging variable in $A$'s $d$th index expression. A *scalar* is either a common scalar variable or an array occurrence with no ranging dimension. A *vector* is an array occurrence with exactly one, a *matrix* with exactly two ranging dimensions. Thus, a diagonal vector is treated as a special matrix.

Let $a$ be an array occurrence. We show now which parameters of its descriptor $d_a$ can be controlled: The *accessed index domain* of a scalar, vector or matrix is defined as the smallest enclosing rectangular polytope of all index positions of ranging dimensions that are accessed in a loop nest. The rectangle is represented

[1] Other induction variables have been transformed away.

```
for i from 1 to 4
  for j from i+1 downto 1
    ... A[2*i][k-j] ...
```

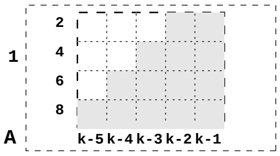



Fig. 3: Descriptor for an example array occurrence. The vertical axis represents the possible values of the first index expression (`i`), the horizontal axis (ascending values towards the right) these of the second (`k-j`). The shaded array elements are accessed.

by the boundaries in each ranging dimension. Lower and upper bound are set such that the stride is always positive or symbolic. The axes are addressed in ascending order: the descriptor should describe *which* array elements are accessed, but not in which order or how often.

In the most interesting case of two ranging dimensions, the accessed index domain is (a subset of) a rectangle. Fig. 3 shows a triangular resp. trapezoidal access shape.

For a trapezoidal access shape, either the *diagonal* (directed from top left to bottom right) or the *counterdiagonal* (vice versa) of the rectangle are defined in a natural way such that (1) there is at least one row and at least one column that are completely accessed, and (2) there is exactly one column or exactly one row or both accessed completely. The diagonal resp. the counterdiagonal divide the rectangle into two trapezoidal sections (Fig. 2(a)) that can be described as combinations of any two neighbouring of the four areas TP (top), LE (left), RI (right) and BM (bottom). To describe the accessed trapezoidal part of a matrix, there are just the corresponding two of the *area bits* TP, LE, RI and BM to be turned on. In the example from Fig. 3, the area bits RI and BM are 1 and the other two are 0. — If all four bits are set, then the matrix is accessed in a rectangular way (*full* access). However, there may be exceptions specified: If an exceptional row index *excidx*[1] is defined, then this row vector is not accessed (Fig. 2(b)). Similarly, *excidx*[2] excludes a column vector, *excdiag*[1] denotes the offset[2] for an excluded diagonal, and similarly *excdiag*[2] for an excluded counterdiagonal. — Finally, if all four bits are zeroed, the *diag* bit indicates whether the diagonal or the counterdiagonal are accessed.

All descriptor parameters (except for the area bits, of course) can be *symbolic* expressions. The descriptor is unique, i.e. if two descriptors are different, then the accessed regions that they describe are also different.

**Operations on descriptors** The PARAMAT pattern recognition tool requires three important operations on descriptors:

*disjdescr*$(d_1, d_2)$ returns TRUE if $d_1$ and $d_2$ can be proved to access disjoint sections of the same array, and FALSE otherwise. We use this as the main data independence test in PARAMAT. The operation is conservative since descriptors always describe a superset of the array elements that are actually accessed. Note that we are only interested in whether $d_1 \cap d_2 = \emptyset$ but not in $d_1 \cap d_2$ itself. Thus, we can avoid computing the intersection explicitly.

[2]The offset is absolute in terms of the vertical axis entries.

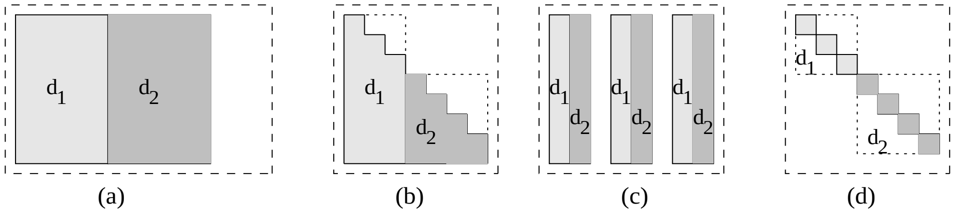



Fig. 4: Some examples for neighboured descriptors.

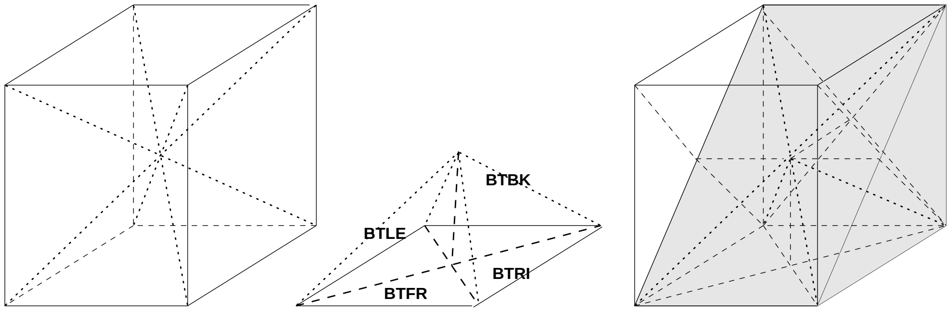



Fig. 5: For three ranging dimensions, the areas move into a fourth of one of the six pyramids separated by the inner diagonals. there are now 24 bits required, like e.g. BTFR for "bottom pyramid, front part". For the prism on the right hand side, all bits of the bottom and the back pyramid are turned on and also the bits LEBM, LEBK, RIBM, RIBK.

$eqdescr(d_1, d_2)$ returns TRUE if $d_1$ and $d_2$ can be proved to be equal, and FALSE otherwise. $geqdescr(d_1, d_2)$ returns TRUE if all array elements described by $d_2$ can be proved to be also described by $d_1$, and FALSE otherwise. These operations are required for computing data flow edges as described in the next section.

$neighbdescr(d_1, d_2)$ returns TRUE if $d_1$ and $d_2$ can be proved to be neighboured array sections, and FALSE otherwise. Two disjoint descriptors $d_1$ and $d_2$ of the same $d$-dimensional array are *neighboured* if (1) their non-ranging index expressions equal, (2) the ranges of $d_1$ and $d_2$ are neighboured for exactly one ranging dimension (for a diagonal matrix: two ranging dimensions), and (3) for all other ranging dimensions, the ranges fit together as shown in Fig. 4, i.e. have the same strides, share at least one bound, and the other bounds match according to the situation. Neighbourhood of descriptors is also of great importance for computing special data flow edges, e.g. to recognize unrolled loops.

**Extending the area bit concept to higher dimensions** Non-rectangular array accesses with more than two ranging dimensions require an extension of the area bit concept to higher dimensions. This is, at least in principle, straightforward. For the three–dimensional case, we can still visualize graphically (Fig. 5) how the four inner diagonals (defined similar to the 2D case) divide the rectangular cube into six pyramids, each of which consists of four parts that are separated by the face diagonals (as already known from the 2D case), thus 24 bits will suffice to code these parts. We can assemble them to describe prisms, pyramids, diagonal faces, or inner diagonal vectors. Since there occur no patterns so far in our library that would handle objects with more than two ranging dimensions, we did not implement this extension up to now.

**Discussion** Our descriptor is in some aspects similar to that described in [BK89]. It is is simpler in that it only allows trapezoidal access structures. We however claim that "parallelogram" or "octogonal" access structures very rarely occur in real programs[3]. In fact, they do not occur in our pattern library at all. On the other hand, our descriptor allows to specify exceptional row, column or diagonal vectors in matrices which is not supported by [BK89]. It further allows non-unit, even symbolic, loop strides and symbolic linear access functions in each dimension. The expensive union and intersection operations from [BK89] are not required for our system: union is handled explicitly by the pattern recognition tool, and regarding intersection, we are only interested in the fact whether the intersection of two descriptors is (provably) empty or not, but not how it does look like. Furthermore, we do not need any information about how often or in which order the data is accessed. The area bit concept provides great compactness of access representation. Due to the area bits, descriptors can be compared very fast.

# Computing Cross Edges

Cross edges are caused by data dependence. They connect parts of the code in the syntax tree that belong to the same thread of computation, even if they are separated textually. They are of crucial importance for pattern recognition.

To compute cross edges properly, we need array data flow analysis to be as exact as possible. Although exact array data flow analysis is generally a very hard problem [Fea91, MAL93], it is highly simplified here since our patterns access arrays in a very regular manner.

We compute descriptors and array data flow immediately before they are needed for the computation of the cross edges, since then maximal data access information is supplied by the pattern instances, and only one loop level has to be considered at a time. For non-matched statements, worst case assumptions must be made (the whole array may be accessed); for matched statements, the access information supplied by the descriptor is tight if some conditions (see above) are met.

Each slot of a pattern instance has one of four access modes: I (ignore), R (read), W (write), RW (read and write). R-slots are upwards–exposed, W-slots are downwards–exposed, and RW-slots are both. Let $I_1$, $I_2$ belong to the same block, $I_1$ preceding $I_2$. There are five different types of *cross edges*:

FLOW: If $I_1$ writes $d_1$ and $I_2$ reads $d_2$ and $geqdescr(d_1, d_2)$ holds and all descriptors $d_3$ of the same array written by instances between $I_1$ and $I_2$ fulfill $disjdescr(d_3, d_2)$, then a cross edge of type FLOW is drawn from $I_1$ to $I_2$. It corresponds to a loop-independent data flow dependence from $I_1$ to $I_2$.

[3] Since we do not need the intersection and union operations for descriptors, this source of non-trapezoidal access shapes is also eliminated.

ANTI: If $I_1$ reads $d_1$ and $I_2$ writes $d_2$ and $geqdescr(d_1, d_2)$ holds and all descriptors $d_3$of the same array written by instances between $I_1$ and $I_2$ fulfill $disjdescr(d_3, d_2)$, then a cross edge of type ANTI is drawn from $I_1$ to $I_2$. This corresponds to a loop-independent anti-dependence from $I_1$ to $I_2$.

INPUT: If $I_1$ reads $d_1$ and $I_2$ reads $d_2$ and $eqdescr(d_1, d_2)$ holds and all descriptors $d_3$ of the same array written by instances between $I_1$ and $I_2$ fulfill $disjdescr(d_3, d_1)$ (and thus also $disjdescr(d_3, d_2)$), then a cross edge of type INPUT is drawn from $I_1$ to $I_2$.

DISJOUT: If $I_1$ writes $d_1$ and $I_2$ writes $d_2$ and $neighbdescr(d_1, d_2)$ holds and all descriptors $d_3$ of the same array written by instances between $I_1$ and $I_2$ fulfill $disjdescr(d_3, d_1)$ and $disjdescr(d_3, d_2)$, then a cross edge of type DISJOUT is drawn from $I_1$ to $I_2$.

DISJIN: If $I_1$ reads $d_1$ and $I_2$ reads $d_2$ and $neighbdescr(d_1, d_2)$ holds and all descriptors $d_3$ of the same array written by instances between $I_1$ and $I_2$ fulfill $disjdescr(d_3, d_1)$ and $disjdescr(d_3, d_2)$, then a cross edge of type DISJIN is drawn from $I_1$ to $I_2$.

In general, the cross edges of a block of statements form a directed acyclic graph. Pattern recognition along a cross edge is guided by the type of the cross edge and the pattern names of the instances involved. If there are several cross edges coming into a pattern instance $I_2$, all candidates $I_1$ are tested successively. As in the vertical case, cross matching is deterministic such that there is always at most one matching template.

If a template for cross matching matches, then $I_1$ and $I_2$ are contracted and $I_2$ is replaced by the new pattern's instance. If $d_1$ and $d_2$ are not completely equal then a residual instance of $I_1$ must remain, as in the following example:

```
I1: VINIT(j=[1:u2], y[i][1:u2], 0.0);
I2: V2CONV(j=[1:u2-1],l,k,y[i][1:u2-1],y[i][1:u2-1],-a[:][:],x[i-:][:-:]);
```

can be matched along the FLOW cross edge caused by `y[i][1:u2-1]`. It remains

```
I1': SINIT(y[i][u2], 0.0);
I2': V2CONV(j=[1:u2-1], l,k, y[i][:], 0.0, -a[:][:], x[i-:][:-:]);
```

# Example: Loop Rerolling

Former machine–specific optimizations, such as redundant IF conditions, loop unrolling, statement reordering and expression reordering, 'encrypt' the meaning of the code. Many of these transformations can be undone on-the-fly at low expense in the pattern recognition phase.

*Loop Rerolling* undoes loop unrolling that may appear (1) as replication at the expression level, (2) as replication at the statement level, or (3) as blocking at the loop level, also known as strip mining (one-dimensional) or tiling (more-dimensional). Loop rerolling and unblocking is, for instance, of crucial importance when matching the Level 1 BLAS routines that are almost all unrolled in some way. As an example, consider the following `daxpy` routine (after the preprocessing transformations)[4] that is 4-way-unrolled at the statement level:

[4]As `n` is symbolic, `mod(n,4)` is treated as a symbolic expression.

```
for (j=1; j<=mod(n,4); j++)  dy(j) = dy(j) + da*dx(j)
for (i=mod(n,4)+1; i<=n; i+=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); }
```

To recognize this as an occurrence of the VAADDSV$^{(1)}$ pattern, we must reroll the loop. Loop distribution and "vertical" pattern matching generate

```
I1:   VAADDSV(j=[1:mod(n,4)], dy,da,dx)
I2:   VAADDSV(i=[mod(n,4)+1:n:4], dy,da,dx)
I3:   VAADDSV(i=[mod(n,4)+2:n:4], dy,da,dx)
I4:   VAADDSV(i=[mod(n,4)+3:n:4], dy,da,dx)
I5:   VAADDSV(i=[mod(n,4)+4:n:4], dy,da,dx)
```

For each slot of each pattern instance, a descriptor is computed and the cross edges are computed as outlined in the previous section. There are now four cross edges of type DISJOUT: from I2 to I3 and I5, from I3 to I4, and from I4 to I5. These cross edges show the neighbourhood relations and guide the pattern recognition process to use the appropriate template that collects all instances in this connected component (I2,I3,I4,I5), merges[5] them and produces

```
I1:   VAADDSV(j=[1:mod(n,4)], dy,da,dx)
I2':  VAADDSV(i'=[mod(n,4)+1:n], dy,da,dx)
```

For the new instance I2', we again compute descriptors and now obtain a new cross edge of type DISJOUT from I1 to I2'. Merging finally yields `VAADDSV(j'=[1:n], dy,da,dx)`.

# Status of Implementation

A prototype of the pattern recognition tool 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[6], including loop rerolling and identification of difference stars. 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.

A block diagram of the Pattern Recognition Tool is given in Fig. 6(a).

# Knowledge–Based Code Generation

The matched intermediate representation is machine independent and opens access to very sophisticated program transformations.

[5]Merging would not be prohibited by other statements that were textually located between these instances and would not write to `dy[mod(n,4)+1:n]`.

[6]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.

Instances of recognized patterns can now be replaced by their best known parallel implementation. There exists at least one for each pattern. These implementations are machine dependent and are parameterized in problem size and data distributions.

As the recognized pattern's names are available, we can now access mathematical background information, e.g., on convergency properties to replace, 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.

This is a first important step away from the classical SPMD semiautomatic code generation method [Ger89] that only inserts masks and communication statements but leaves the original computation structure unchanged. It is now also no problem to locally get rid of the owner-computes rule.

Local algorithm replacement also enables re-introducing machine-specific code transformations such as loop blocking that may have been removed by the loop rerolling technique described above. Now the compiler itself is free to choose a suitable blocking scheme for optimizing vector register allocation [KPR92] or cache utilization [WL91].

If, for a given pattern instance, the problem size is known at compile time and if it is small, PARAMAT may also decide to prohibit parallelization if sequential execution will be faster, thus avoiding speed-down of the target program. If the problem size is not known at compile time, a suitable run time test should be inserted into the generated code. All these optimizations are hidden for the user.

Each pattern implementation accesses data in an individual manner. Thus, for each pattern implementation, PARAMAT knows (at least) one favorite alignment (to minimize communication) and one favorite distribution (to maximize parallelism) of all the arrays which this pattern is operating on.

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. To help with the combinatorial complexity, we make use of our favorite-partitioning knowledge as starting configurations when performing a global search for the optimal data distribution, e.g. by branch and bound, as in [DHR93]. [DHR93] also enables *static array redistribution* which is also a NP–complete problem [Kre93].

For each pattern implementation, PARAMAT can inspect a table of run times — again parameterized in problem size and data distribution — that have been measured at an earlier time on the target machine. 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, overlapping of computation and communication etc.) better than theoretical, idealized estimation functions.

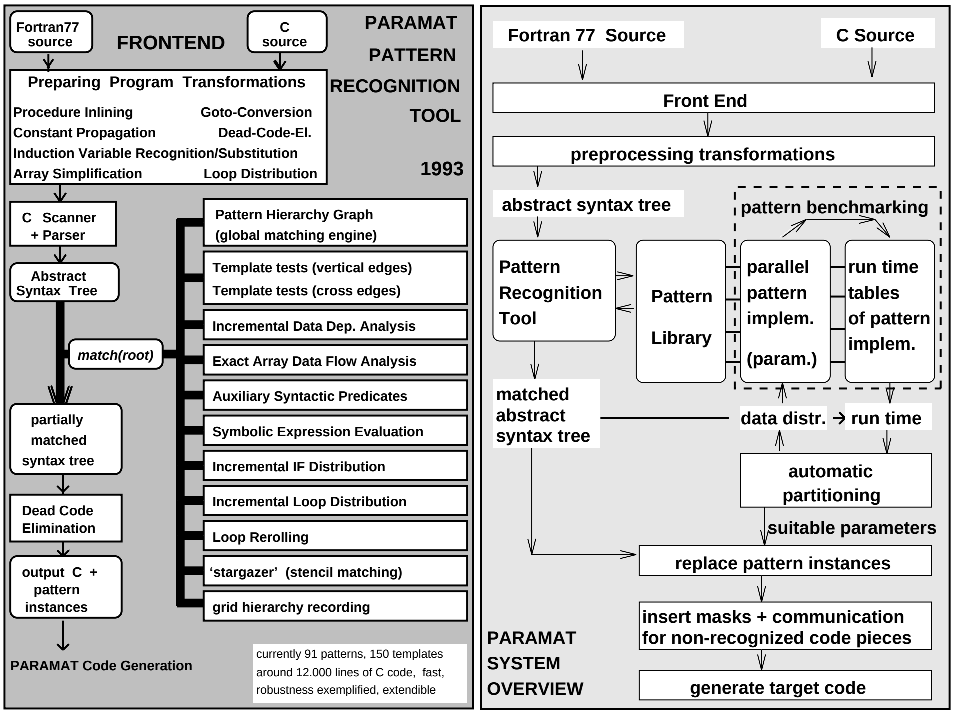



Fig. 6: (a) PARAMAT Pattern Recognition Tool; (b) PARAMAT system design

Fig. 6(b) summarizes the PARAMAT system design.

# References


[BK89] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 41–53, 1989.

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

[DHR93] 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, 1993.

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

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

[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.

[HPF93] High Performance Fortran Forum HPFF. High Performance Fortran Language Specification, Final Version 1.0, May 1993.

[Keß93a] Christoph W. Keßler. Knowledge–Based Automatic Parallelization by Pattern Recognition. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code*

*Generation, Data Distribution and Performance Prediction*, pages 110–135. Verlag Vieweg, 1993.

[Keß93b] Christoph W. Keßler. Pattern Recognition Enables Automatic Parallelization of Numerical Codes. In *Proc. of the Fourth Int. Workshop on Compilers for Parallel Computers, Delft, NL*, Dec. 1993.

[KP93] Christoph W. Keßler and Wolfgang J. Paul. Automatic Parallelization by Pattern Matching. In *Proc. of Second Int. Conference of the Austrian Center for Parallel Computation*, pages 166–181, Oct. 1993.

[KPR92] 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*, pages 77–91. Springer Workshops in Computing Series, 1992.

[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.

[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.

[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.

[WL91] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 30–44, 1991.