Pattern Recognition Enables Automatic Parallelization of Numerical Codes

Christoph W. Keßler*

Abstract

We describe the design of the PARAMAT system which will be able to automatically parallelize many numerical application codes without any user interaction, for execution on distributed memory message-passing multiprocessors (e.g. iPSC/860, CM-5, nCUBE-2, ...).

The key idea is a high-level pattern recognition approach which permits local restoration of program concepts of a wide class of scientific codes by identifying typical programming patterns. This is also applicable to dusty deck codes that may be encrypted by former machine-specific code transformations.

The presented pattern recognition tool is fast and robust against many common transformations such as loop distribution, loop interchange, loop blocking and loop unrolling.

Successful pattern recognition enables sophisticated code transformations including algorithm replacement. It also guides data alignment and distribution and allows more exact prediction of the performance of the parallelized target code than usually possible.

1 Automatic Parallelization by Pattern Recognition

In this paper, we briefly present the PARAMAT system, a knowledge–based automatic parallelization system. It is applicable to a large class of scientific programs written in C, Fortran77, Fortran 90 or similar languages for execution on distributed memory MIMD message passing systems (DMS).

These DMS machines are well–known to be difficult to program. The most important reasons are the following:

- Parallel code must contain explicit message passing statements, and programming explicit message passing is complex, tedious and error-prone.
- 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.
- 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.
- Run time prediction for nontrivial codes on real machines is a very problematic matter for reasons of network contention, message protocols, undocumented hardware features, and others.

Message passing statements can be generated automatically today by *semiautomatic* parallelization. The user has to provide data distribution and optimizing transformations, either in the form of interactive commands in tools like SUPERB ([ZBG88, Ger89, GZ90]) or MIMDizer ([SWW91]), or in the form of language constructs or compiler directives in an explicitly parallel language such as Fortran-D ([HKT91], Vienna Fortran ([CMZ92]), HPF (High Performance Fortran [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

*Graduiertenkolleg Informatik, Universität Saarbrücken, email: kessler@cs.uni-sb.de

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* ([Keß93]).

From this observation, we are able to construct an automatic parallelizer with the following key ideas:

- First, there must be a pattern recognition tool that works fast and reliably. Recognized patterns will be replaced by a pattern *instance*, looking similar to a Fortran90 intrinsic function call.
- Once we know what the program locally does, we can infer a lot of additional knowledge on mathematical properties, efficient implementations, favorable data distributions and run time behaviour.
- The parallelization system can then easily use this knowledge to guide a sophisticated parallelization process.

The rest of this paper is organized as follows: The next section motivates pattern recognition by analyzing several scientific programs. Section 3 briefly introduces a tool that is able to reliably recognize these patterns. Sections 4, 5 and 6 show how the recovered semantic information is used to select suitable implementations and data distributions, and to support more exact performance prediction. The last sections summarize the overall automatic parallelization system called PARAMAT ("PARallelize Automatically by pattern MATching"), discuss its advantages and disadvantages, and outline future directions of research.

2 Patterns in Scientific Programs

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 [Mül89] or the algorithms occurring in a parallel numerics course [Lou92]: Basic linear algebra subroutines (see also [LHKK79, DDHH88]), 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]. These algorithms are the basic building blocks of many numerical applications.

We observed that these numerical algorithms are made up of only a few (around 150) characteristic programming schemes (called *patterns*) such as e.g. vector and matrix operations, simple recurrences, relaxation operations, simple reduction operations, convolutions, FFT and so on.

Some of the patterns that we have currently collected in the Basic PARAMAT Library ([Keß93]) are summarized in Tab. 1.

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 exemplify this by examining the source codes of standard numerical benchmark programs, e.g.:

- the *Livermore Loops*, 24 inner loops from numerical application programs that represent the most time-consuming program kernels;
- some kernels from the NAS benchmark program ([BB86]),
- the *Purdue Set*, 14 kernels from the HPF benchmark suite ([MFL$^+$92])

and of actual application programs, e.g., selected codes from the Perfect Club Benchmarks (especially from the programs FLO52, SPEC77 and DYFESM, see also [Ber92]), or multigrid programs.

With the current version of the Basic Pattern Library — containing only 150 patterns, 68 of them already implemented — we are able to cover *completely* the basic numerical codes mentioned above; the results for the Livermore Loops are given in Tab. 2 and for the Purdue Set in Tab. 3.

order	patterns	number
0	scalar arithmetics, init, max, min, swap, read, write	20
	`MULTIADD`, `MULTIMUL`, `STAR` (grid stencil 1D and 2D)	4
1	vector instructions, v-init/-assign/-copy/-swap, v-read/write, SV	16
	reductions: v-sum, scalarproducts, vector norms	9
	reductions: vector maximization/minimizations	6
	1D relaxation steps (Jacobi, Gauss–Seidel)	2
	first order linear recurrences	2
	intermediate forms of 1D convolution	2
	global vector update, pivoting	4
2	matrix operations, m-init/-assign/-copy, m-read/-write, SM	12
	vector/matrix multiplication patterns	4
	2D-reductions: m-sum, concurrent v-sum/prod, matrix norms	6
	2D-reductions: m-max/min, row/col-max/min	8
	2D relaxation steps (Jacobi, Gauss–Seidel, ...)	4
	global matrix updates, triangular solver	3
	1D convolution; intermediate forms of 2D convolution	4
3	matrix multiplication	2
	matrix inversion, Gaussian elimination	4
	intermediate forms of 2D convolution	2
4	2D convolution	1

Table 1: A brief summary of the patterns included into the current version of the Basic PARAMAT Pattern Library. The number of templates per pattern rarely exceeds 3; most patterns have only one or two. All BLAS routines operating on dense real matrices are contained.

Loop	Name	recognized patterns	recognized loops
1	Hydrofragment	`GVOP`	1 from 1
3	Inner Product	`SSP`	1 from 1
5	tri-diag. elim., below diagonal	`FOLR`	1 from 1
8	A.D.I Integration	`VDIF` (3), `GVOP` (3)	6 from 6
9	Numerical Integration	`GVOP`	1 from 1
10	Numerical Differentiation	`GVOP` (19)	19 from 19
11	First Sum	`FOLR`	1 from 1
12	First Difference	`VDIF`	1 from 1
13	2D particle in a cell	`VCOPY` (4), `VADD` (2),`GVOP` (4)	10 from 17
14	1D particle in a cell	`GVOP` (3), `VCOPY`, `VADD` (2)	6 from 12
18	2D explicit hydrodyn. fragment	`MAADDSM` (2x)	2 from 6
21	Matrix Product	`MMO`	1 from 1
22	Planckian Distribution	`GVOP` (2)	2 from 2
23	2D implicit hydrodyn. fragment	`MGAUSSSEIDEL`	1 from 1
24	1D Minimization	`VMINLOC`	1 from 1

Table 2: Analysis of the Livermore Loops: currently recognizable patterns. The right–hand column indicates how many loops (after applying loop distribution) can be covered by patterns from the Library. `GVOP` denotes a general vector operation, `FOLR` first order linear recurrences, `VDIF` a 1D difference star; `VMINLOC` finds in a vector the location with minimal absolute value.

It is clearly no problem to include Fortran 90's array features and intrinsic array manipulation functions. These just occur as additional templates of their corresponding PARAMAT patterns.

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.

[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).

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

Table 3: Analysis of the Purdue Set: currently recognizable patterns. The right–hand column indicates how many loops (after applying loop distribution) can be covered by patterns from the Library. `VINITSP/MINITSP` denote vector/matrix initializations by special functions depending on the loop variables, `VINC` increments a vector by a constant, and `FSUM` adds up subsequent values of a scalar function.

3 Pattern Recognition

We use a deterministic algorithm similar to bottom-up pattern matching on the source program's abstract syntax tree - this is very fast since every node of the syntax tree is considered only once, and because the Basic Library of around 150 patterns makes effective use of the hierarchy of patterns.

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.

In order to be able do 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. ([Keß94]).

3.1 Pattern Hierarchy Graph

The pattern library is hierarchically organized. We represent it as a directed acyclic[2] 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. For instance, there is an edge from SSP (inner vector product) to VM (vector-matrix-multiplication), or from SINIT (scalar initialization) to VINIT (vector initialization). The PHG edges are used to trigger pattern matching.

If all possible subpatterns and their ancestors are present in the PHG, then the PHG is called complete. The pattern hierarchy graph of Matrix-Vector multiplication is a subgraph of the complete PHG of Matrix-Matrix-Multiplication (see Fig. 1).

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.

[2] For the reason of improved performance, self-cycles from a pattern to itself are admitted.

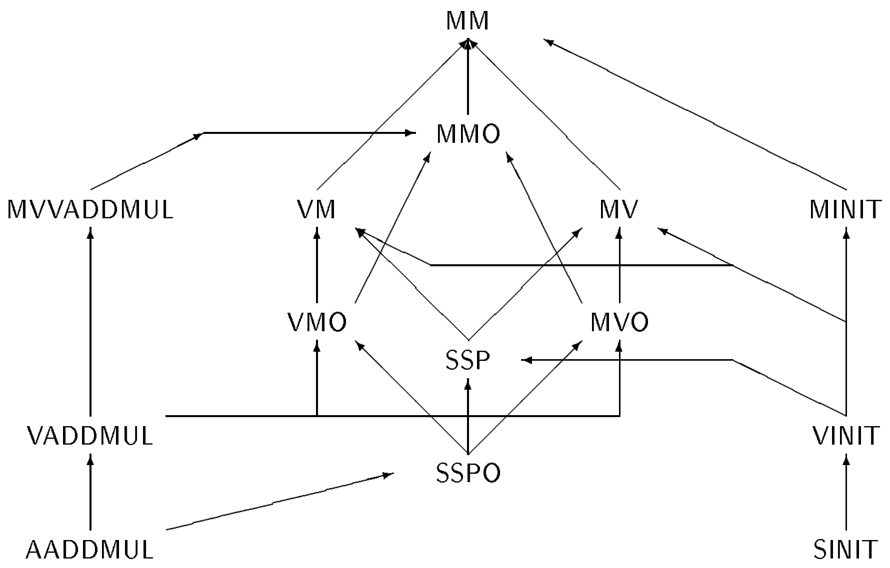


Figure 1: The pattern hierarchy graph of Matrix Multiplication. It covers all possible (sensible) ways how matrix multiplication may be coded. Handling of temporary variables is also possible.

3.2 Pattern Recognition Algorithm

Our pattern recognition algorithm is similar to bottom–up pattern matching as used in automatically generated code generators (e.g., [FHP92, CC92, Wil79]), but works deterministically, and enables (nearly) arbitrary local deviations from the standard matching scheme to save patterns and matching time. This offers the high degree of flexibility that is necessary here and that cannot be supplied by current tree pattern matching kits.

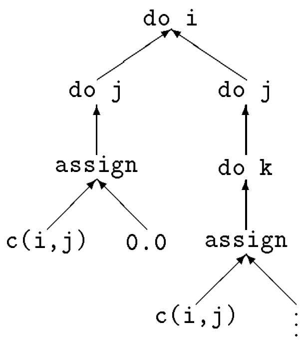
The source program's syntax tree is traversed bottom-up and from the left to the right. Recognized nodes are replaced by a pattern instance. Each node is visited only once, and for each node, there are only a few possible patterns due to the PHG structure. Thus, matching time is guaranteed to be linear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

3.3 Standard Pattern Matching: A simple example

Let us illustrate this procedure with a simple example. Matrix multiplication is well suited for this purpose since it is no trivial example but is not made up of too many subpatterns so that its pattern hierarchy graph (Fig. 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 statement S1 will be replaced by the pattern instance `SINIT(c(i,j),0.0)` since this is the leftmost leaf of the syntax tree.

Pursuing this branch further towards the root, the `do j` loop around S1 will be matched with the `SINIT` instance resulting in a new pattern instance `VINIT(j,c(i,:),0.0)` (vector initialization). The slot entry for the loop variable `j` in this instance also summarizes the bounds of the `do j` loop.

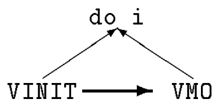
Then the algorithm descends further towards S2, replaces the assignment statement leaf corresponding to S2 by a pattern instance `AADDMUL(c(i,j),a(i,k),b(k,j))` (accumulatively adding products) and, following the suitable edge in the PHG, merges this with the `do k` loop giving the instance `SSPO(k, c(i,j), a(i,:), b(:,j))` (non-initialized standard inner product). From the pattern hierarchie graph (Fig. 1) we know all possible superpatterns for SSPO. We try them successively and find out that `VMO(j,k, c(i,:), a(i,:), b(:,:)` (vector-matrix multiplication) matches the `j` loop.

The complete information of the original source program is now concentrated in the names and the slot entries of the pattern instances. All data accesses (in particular, for vectors and matrices) are summarized by a *descriptor* associated with each slot.

At this point, it is exactly known by the descriptors which data is read and written by these two pattern instances. We can thus perform exact array data flow analysis which yields that in the `VINIT` instance a row vector of `c` is written that is read and overwritten in the `VMO` instance.

As the result of this array flow analysis, we draw a *cross edge* in the syntax tree from the `VINIT` instance to the `VMO` instance. This cross edge is caused by `c(i,:)` and of type FLOW.

```
do i=1,n
   VINIT(j, c(i,:), 0.0)
   VMO(j, k, c(i,:), a(i,:), b(:,:))
enddo
```



As we can see from the PHG (see Fig. 1) and the matching rules, these two instances can be merged along the cross edge from VMO to VM. This results in a new instance `VM(j,k, c(i,:), a(i,:), b(:,:))`, a vector-matrix multiplication with initialization. This, in turn, can further be matched with the `do i` loop into a MM instance (Matrix-Matrix-Multiplication).

3.4 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. Some examples can be found in [Keß94].

To compute cross edges properly, we need data dependency and exact array data flow analysis, as stated in the example above. Although exact array data flow analysis is generally a very hard problem (see e.g. [Fea91, MAL93]), it is highly simplified here. For each slot of a pattern instance, we compute a compact *descriptor*[3] that exactly summarizes which data are accessed (e.g., triangular parts of a matrix, excluded line/row vectors etc.) by that instance. Descriptors can be compared very fast. We compute descriptors and array data flow just 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.

Besides type FLOW (emanating from a data flow dependence), cross edges may also be of type ANTI (due to anti dependence), INPUT (both instances read the same object), DISJOUT (both instances write disjoint sections of the same array) or DISJIN (both instances read disjoint sections of the same array). In general, the cross edges of a block statement form a directed acyclic graph.

3.5 Status of Implementation

A block diagram of the Pattern Recognition Tool is given in Fig. 2.

The current implementation reliably recognizes 68 patterns with 117 templates. The template matching functions are sequences of constraints written in C. Writing them down has been made handy by the availability of useful syntactic predicates. The system is able to work on symbolic expressions.

[3] This descriptor is simpler than that of [BK89] but well-suited for our patterns.

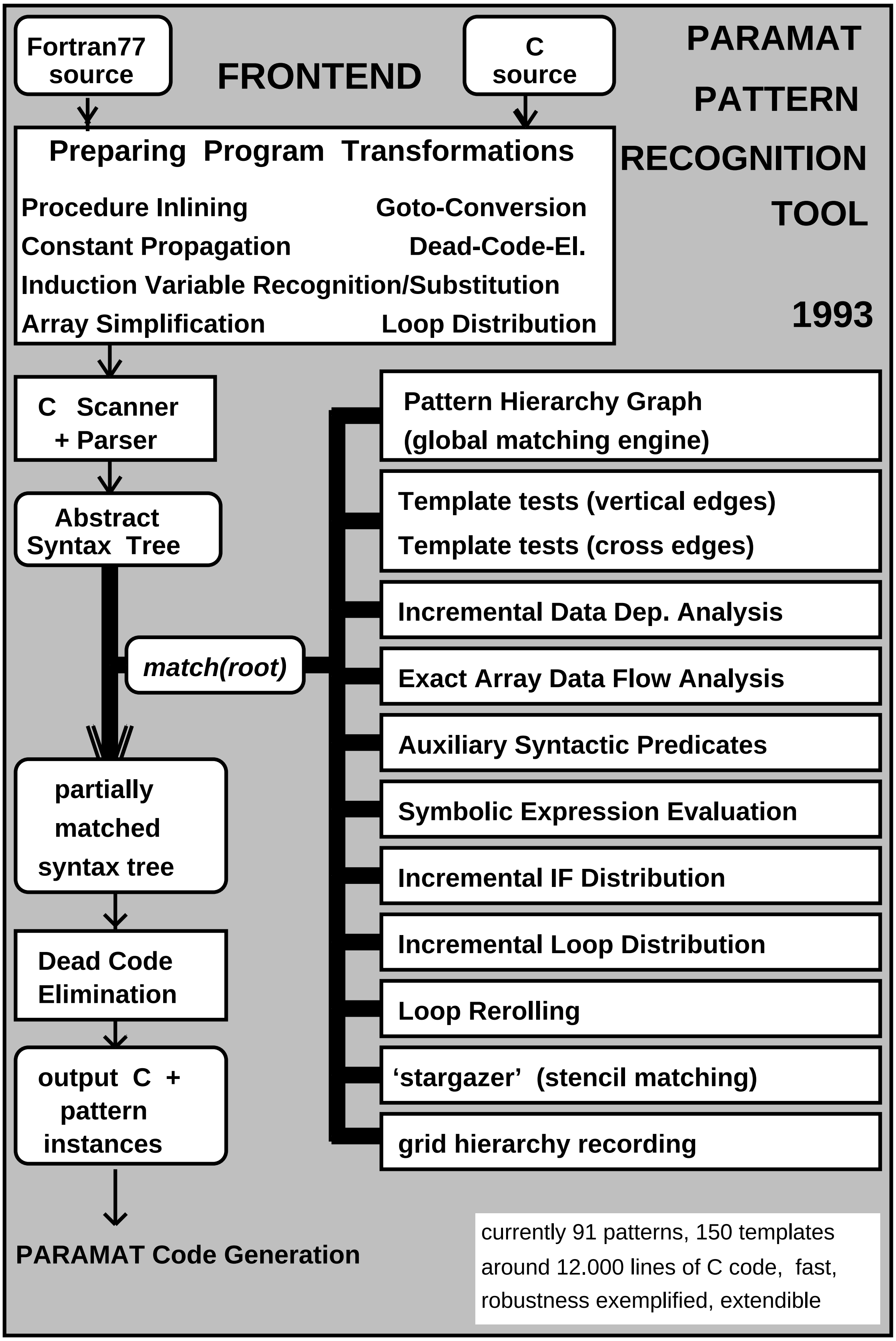


Figure 2: Block diagram of the PARAMAT Pattern Recognition Tool

pattern (instance)	algorithm	alignment recomm.	distribution recomm.
`MCOPY(...,A,B)`	matrix copy	$A \equiv B$	arbitrarily
`VCOPY(...,V,W)`	vector copy	$V \equiv W$	arbitrarily
`MJACOBI(...,A,B,...)`	one Jacobian relax. step	$A \equiv B$	quadratic blocks
`MM(...,C,A,B)`	matrix multiplication	$A \equiv C \lor B \equiv C$	`A` by row, `B` by column
`SSP(...,s,V,W)`	standard inner product	$V \equiv W$	arbitrarily

Table 4: Array alignment and distribution (dep. on target) recommendations for some patterns

Furthermore, the Pattern Recognition Tool is able to distribute, to reroll and unblock `do` loops, to distribute IF statements, to remove redundant IF conditions, to detect various stencils (difference stars) and to record possible grid hierarchies in multigrid programs (see [Keß94]).

4 Algorithm Replacement and High-Level Transformations

Instances of recognized patterns are now replaced by their best known parallel implementation. There exists at least one for each pattern. These implementations are, of course, machine dependent and are parameterized in problem size and data distributions (cf. Section 5).

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 (e.g., from Livermore Loop 23) 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.

For recognized Linear Recurrences (e.g. `FOLR`, see [Keß94]), we also can apply a suitable number of recursive doubling steps ([KS73]).

This is a first important step away from the classical SPMD semiautomatic code generation method (e.g., [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.

Furthermore, we can make optimal use of predefined routines that are already supplied with the runtime environment of the target machine, e.g. for global sum (`VSUM`), global product (`VPROD`), global OR etc.

Local algorithm replacement also enables sophisticated, target–machine specific code transformations, such as loop blocking for optimizing vector register allocation ([KPR92]) or cache utilization ([WL91]). These optimizations are hidden for the user. It is also no problem to re-introduce here optimizations that have been removed before or within the pattern recognition phase (e.g., introduction of temporary variables, loop unrolling, etc.).

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.

5 Automatic Data Distribution

Each pattern implementation accesses data in an individual manner. Thus, for each pattern implementation, there is (at least) one favorite alignment (to minimize communication) and one favorite distribution (to maximize parallelism) of all the arrays which this pattern is operating with. Tab. 4 gives some examples.

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 (cf. [LC90]), thus automatic partitioning may take exponential time in the worst case.

A truly automatic parallelization system must be able to determine — at least, approximately — good data distributions automatically, i.e., without any hint by the user. A reasonable approach seems to be a branch-and-bound heuristic as in [DHR94]), provided that the number of distribution alternatives is limited severely to maintain acceptable run time. For matrices, for instance, we allow only row-/column-/block-contiguous and row- or column–cyclic distributions (cf. e.g. [GB90]), since for all the applications considered so far these are enough to supply good distributions. Furthermore, we limit the number of partitionable array dimensions to 2. We use the distribution recommendations for the patterns as start configuration when searching for good data layout. Together with the simplification of the source program by pattern matching, these restrictions make the application of a branch–and–bound search, as done in [DHR94], computationally feasible.

This approach ([DHR94]) also enables *static array redistribution*. The boundaries between different pattern instances in the matched source program are natural places for static array redistribution. Another algorithm for static array redistribution has been proposed in [CKKM94] and also fits well into our framework.

6 Automatic Parallel Run Time Prediction

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 "bottom-up" estimation functions.

7 PARAMAT system design

Consider the block diagram of the PARAMAT system in Fig. 3. The core of PARAMAT is the pattern recognition tool (cf. Fig.2) with the pattern library (cf. Fig. 1). For each pattern there exists at least one suitable parallel implementation on the target machine, parameterized in problem size and data distributions. Each implementation, in turn, has been benchmarked earlier, resulting in parameterized run time function tables which can be inspected by the run time estimator of the data distribution search algorithm. If the data distribution has been fixed, the pattern instances are replaced by the corresponding implementation. Then there only remains some residual adaptation (masks and communication insertion) for not recognized code portions, and finally, everything is translated to machine language.

8 PARAMAT as Fortran 77 to HPF converter?

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

9 Conclusions

The PARAMAT approach to automatic parallelization consists of three basic ideas: First, we observe that we can cover large parts of many numerical codes by a small set of typical programming patterns. Second, we give a recognition algorithm similar to bottom-up pattern matching which tries to locally

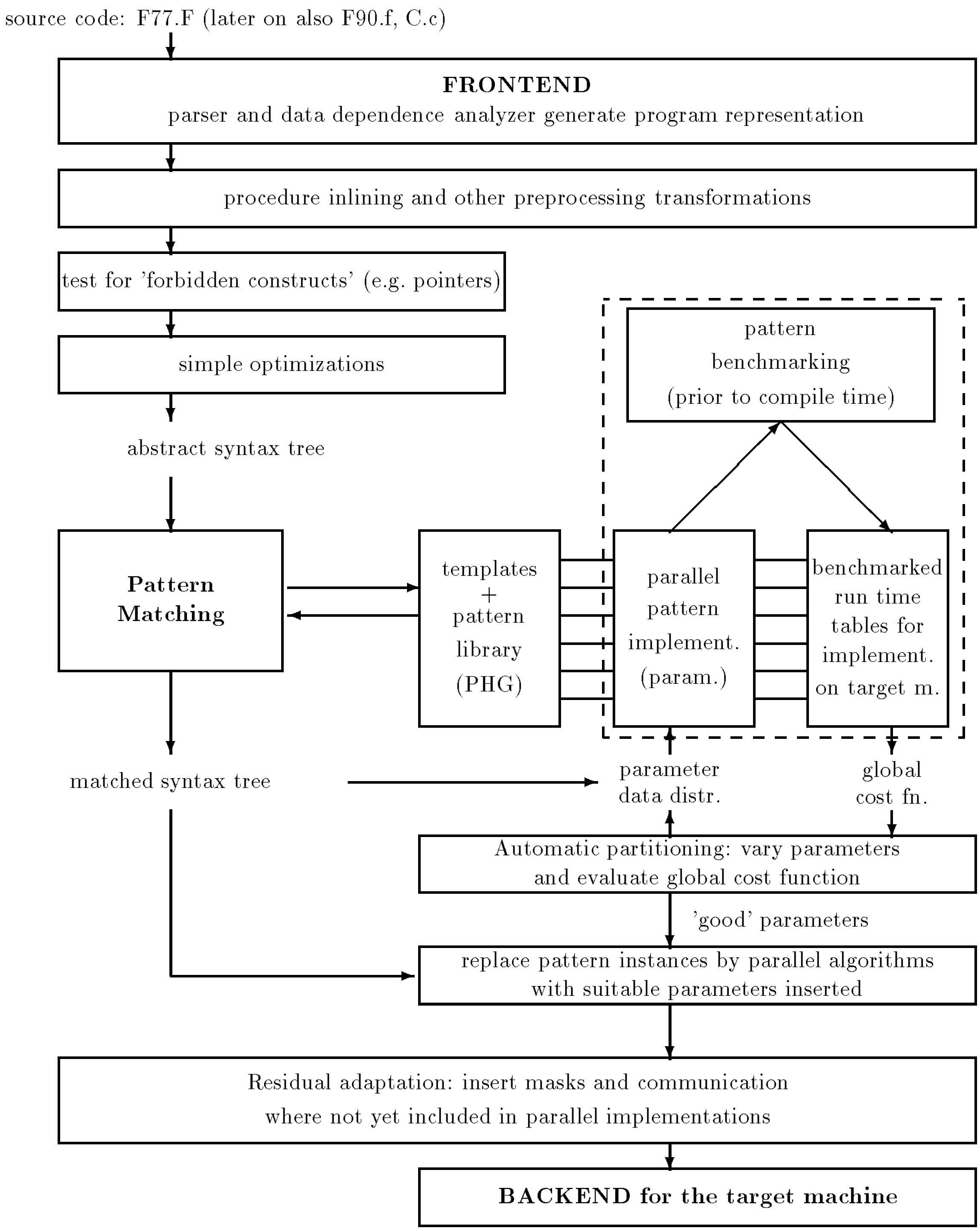


Figure 3: Block diagram of the overall PARAMAT system. Only the components enclosed in the dashed rectangle (and the back end, of course) are dependent on the target machine. Parallel algorithms for each pattern and their run–time approximations, parameterized by the data partitionings of the arrays involved in the resp. pattern instance, have to be precomputed once for each hardware constellation (pattern benchmarking) *before* compile time.

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 sophisticated optimizing code transformations including algorithm replacement.

Pattern recognition simplifies the program graph and thus alleviates global optimizations such as determining optimal array alignment and partitioning. Pattern recognition enables safe algorithm replacement and optimal use of the highly optimized operating system routines supplied with the hardware environment. It also supports better performance prediction accuracy by benchmarking higher–level patterns. Beyond automatic parallelization, the pattern recognition tool may be of great use at reengineering sequential dusty–deck codes for porting them to other languages or target machines.

PARAMAT makes the experience of parallel programming and optimization experts accessible to all scientific programmers and thus avoids re-inventing the wheel for each program parallelization project.

Of course, we may get the same effect if the programmer would a priori write his program in a *pattern language*, i.e. he would have to use the predefined patterns from PARAMAT's pattern library already at the coding stage, thus just bypassing the pattern recognition tool. This procedure is possible in the PARAMAT system since it understands given pattern instances in the source program as well; however, this again imposes much work and discipline on the programmer, and Fortran 77 is widely accepted to be a standard for implementation of numerical programs that application programmers are not willing to abandon.

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 first target machine will be the Intel iPSC/860. A prototype of the PARAMAT pattern recognition tool is already running. At the university of Kaiserslautern, the PARAMAT pattern recognition tool is used as a front-end for code generation for the Xputer parallel machine ([HKS94]).

The PARAMAT system is open for extensions. The pattern library can be extended by adding more pattern modules according to individual application areas. The computation of the run time approximation functions can be automated by a universal benchmarking tool. Changing the hardware platform means only to load another base of parallel algorithms and their run time functions. Thus the PARAMAT system will always be up to date for the latest hardware environments available.

References

[BB86] David H. Bailey and John T. Barton. The NAS Kernel Benchmark Program. Numerical Aerodynamic Simulations Systems Division, NASA Ames Research Center, June 1986.

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

[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.

[CC92] 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.

[CKKM94] Alan Carle, Ken Kennedy, Ulrich Kremer, and John Mellor–Crummey. Automatic Data Layout for Distributed Memory Machines in the D Programming Environment. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 136–152. Verlag Vieweg, 1994.

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

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

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

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

[FHP92] 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.

[GB90] 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.

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

[GZ90] Michael Gerndt and Hans P. Zima. Superb: Experiences and Future Research. Technical Report ACPC/TR 90-5, Austrian Center for Parallel Computation, October 1990.

[HKS94] Rainer W. Hartenstein, Christoph W. Keßler, and Karin Schmidt. Knowledge–Based Compilation and Restructuring Transformations for Xputers, submitted (1994).

[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.

[Keß93] Christoph W. Keßler. The Basic PARAMAT Pattern Library, 1993.

[Keß94] 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, 1994.

[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.

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

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

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

[Lou92] A. K. Louis. Parallele Numerik. Course script and selected programs, unpublished, Universität Saarbrücken, 1992.

[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.

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

[Mül89] Silvia M. Müller. Die Auswirkung der Startup-Zeit auf die Leistung paralleler Rechner bei numerischen Anwendungen. Master thesis, Universität Saarbrücken, 1989.

[SWW91] R. Sawdayi, G. Wagenbreth, and J. Williamson. MIMDizer: Functional and Data decomposition. In *Compilers and Runtime Software for Scalable Multiprocessors.* Elsevier, 1991.

[Wil79] Reinhard Wilhelm. Computation and Use of Data Flow Information in Optimizing Compilers. *Acta Informatica*, 12:209–225, 1979.

[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.

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