Automatic Parallelization by Pattern–Matching

Christoph W. Keßler*** and Wolfgang J. Paul

Fachbereich Informatik
Universität Saarbrücken
Postfach 1150
D-66041 Saarbrücken, Germany

Abstract. We present the top–down design of a new system which performs automatic parallelization of numerical Fortran 77 or C source programs for execution on distributed–memory message – passing multiprocessors such as e.g. the INTEL iPSC860 or the TMC CM–5.
The key idea is a high–level pattern–matching approach which in some useful way permits partial restructuring of a wide class of numerical programs. With only a few hundred patterns, we will be able to completely match many important numerical algorithms. Together with mathematical background knowledge and parallel compiler engineering experience, this opens access to a new potential for automatic parallelization that has never been exploited before.

1 Introduction and Overview

Current distributed memory multiprocessers are hard to program, and 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 tedious work to an optimizing parallelizing compiler. The programmer wants to feed in just 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 concentrates more on parallel programming languages for distributed memory systems (such as e.g. Fortran–D ([HKT91]) or Vienna Fortran ([CMZ92])) than on parallelizing compilers.

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. in the form of directives, as in Fortran-D), and the job of the compiler is to insert masks and communication statements according to this specification. This 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.

* Research funded by Graduiertenkolleg Informatik at Saarbrücken University
** email: kessler@cs.uni-sb.de

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 of his program in order to decide which distributions to choose 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. Changing the hardware platform means repeating the whole parallelization procedure.

Recently some research has been done on automatic derivation of data distribution ([KLS90, GB90, LC90, Who91, KK91]), 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 matching on a quite low level to facilitate parallelization and optimizations ([GB90, PP91, CHZ91a, CHZ91b]). We claim that pattern matching can be usefully applied to entire programs as the *core* of an 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 [Mül89] or the algorithms occurring in a parallel numerics course [Lou92]. These algorithms contain 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.

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 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 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 implementations (called *templates*) of the same pattern which 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 4.

Once this pattern matching 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.

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, the following transformations[3] should be carried out in order to facilitate pattern matching:

- procedure inlining[4]
- constant propagation
- induction variable recognition (substitution, if possible)
- temporary variable recognition (substitution or temporary scalar expansion)
- dead code elimination
- conversion of GOTO's into if–then–else or while statements, where possible
- loop distribution

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. [SCMB90]). In this work we restrict ourselves to static parallelization.

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.

[3] For a detailed description of optimizing transformations, see e.g. [ZC90].

[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.

Of course we cannot go beyond a certain level — one single pattern for a whole application program of several hundred lines would be nonsense. If a piece of program is too exotic and occurs very seldom, then it is not worth becoming a pattern. Thus the philosophy is to have as many patterns as necessary, and as few as possible.

The basic algorithms considered in [Mül89] and [Lou92] 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. [McM86]),
- some kernels from the NAS Benchmark program (cf. [BB86]),
- a LU decomposition code from the netlib,
- a least–square Conjugate Gradient solver[5] from [Lou92],
- selected codes from the Perfect Club Benchmarks
- 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 [Keß93]. A brief summary of the patterns is given in Tab. 1. Some ideas for the efficient distributed–memory parallel implementation of all the patterns occurring in these algorithms are summarized in [Mül89].

As one can see from Tab. 1, the number of low–level patterns (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 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 100 patterns — we are able to cover *completely* (and thus, to parallelize automatically) a lot of the basic numerical codes from [Mül89] and [Lou92], 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. 2. We obtained similar results for the other application codes listed above.

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

order	patterns	number
0	scalar arithm. operations, init, max, min, swap, assign, read write	20
	difference stars and substars (first and second order)	3
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	2
	first order linear recurrences	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
	diverse relaxation steps	4
	global matrix updates (GJstep, GRstep,..., MSOLVEL/-U)	6
3	matrix multiplication	2
	matrix inversion, GReduce	2

Table 1. A brief summary of the patterns included into the current version of the Basic PARAMAT Pattern Library. No pattern has more than three different templates, most patterns have only one or two. All BLAS routines operating on dense real matrices have been entered.

4 Pattern–Matching

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 the possible patterns. For each template of each pattern there exists a small procedure which tests whether a given piece of program matches this pattern, and if yes, it returns an *instance* of this pattern where the formal pattern parameters are bound to the corresponding actual parameters occurring in the program piece.

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.

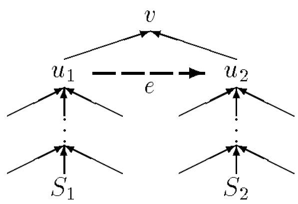
We distinguish between two kinds of directed edges between nodes:

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.

LL	Name	recognized patterns	recogn. loops
1	Hydrofragment	GVOP$^{(1)}$	1 from 1
3	Inner Product	SSP$^{(1)}$	1 from 1
5	tri-diag. elim., below diagonal	FOLRO$^{(1)}$	1 from 1
8	A.D.I Integration	VDIF$^{(1)}$(3x), GVOP$^{(1)}$(3x)	6 from 6
9	Numerical Integration	GVOP$^{(1)}$	1 from 1
10	Numerical Differentiation	GVOP$^{(1)}$	19 from 19
11	First Sum	FOLR$^{(1)}$	1 from 1
12	First Difference	VDIF$^{(1)}$	1 from 1
13	2D particle in a cell	VCOPY$^{(1)}$(4x),VADD$^{(1)}$ (2x),GVOP$^{(1)}$(4x)	10 from 17
14	1D particle in a cell	GVOP$^{(1)}$(3x),VCOPY$^{(1)}$ (1x),VADD$^{(1)}$(2x)	6 from 12
18	2D explicit hydrodyn. fragment	MADDMUL$^{(2)}$(2x)	2 from 6
21	Matrix Product	MMO$^{(3)}$	1 from 1
22	Planckian Distribution	GVOP$^{(1)}$(2x)	2 from 2
23	2D implicit hydrodyn. fragment	GAUSSSEIDEL$^{(2)}$	1 from 1
24	1D Minimization	VMINLOC$^{(1)}$	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$^{(1)}$ denotes a general vector operation, FOLRO$^{(1)}$ and FOLR$^{(1)}$ first order linear recurrences, VDIF$^{(1)}$ a 1D difference star; VMINLOC$^{(1)}$ finds in a vector the location with minimal absolute value.

2. *horizontal 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[6] data dependence from a statement S_1 to another statement S_2, then a horizontal 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:



[6] 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 2 before.

Then the horizontal 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 horizontal edges following the textual order of the statements in the source program.

Thus the vertical edges form a tree while the horizontal edges form a directed acyclic graph on each control hierarchy level. For an example, see section 4.4.

The horizontal 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.

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. 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 $order(p_1) \leq order(p_2)$.

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.

4.3 The Matching Algorithm

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**
/$\star$ now all vertical predecessors (substmt's, subexpr's) of *node* are known $\star$/
/$\star$ possible patterns p of *node* are all direct PHG successors (superpatterns) of the patterns already computed for the sons s of *node* $\star$/

```
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 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.
```

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

This pattern matcher is similar to other bottom–up pattern matching algorithms such as used in automatically generated code generators, e.g. BURG ([FHP92]). These systems automatically[7] generate a tree automaton whose state table corresponds to our PHG. Unfortunately, all automatic systems work only on (syntax) trees; this disables matching along horizontal 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 did for matching difference stars (see [Keß93]).

4.4 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. 1) remains quite handy.

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

[7] 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.

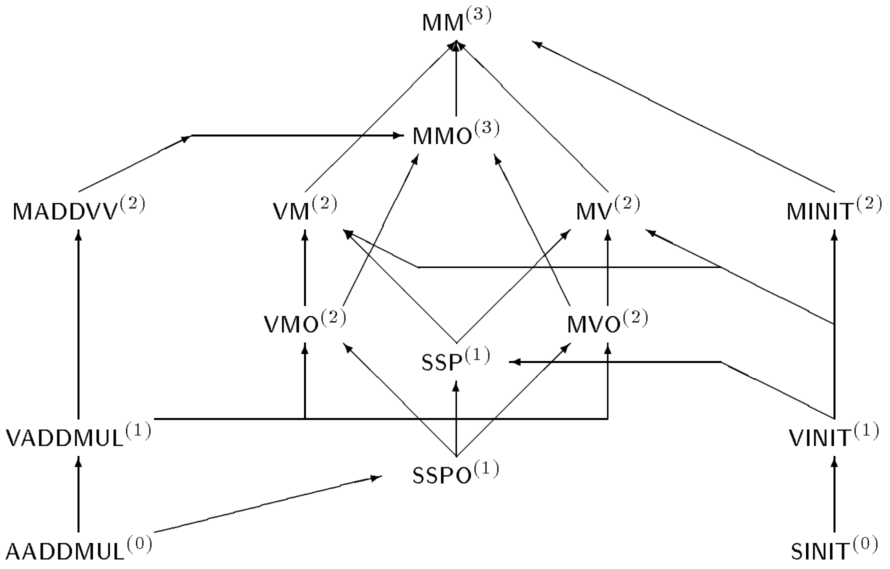
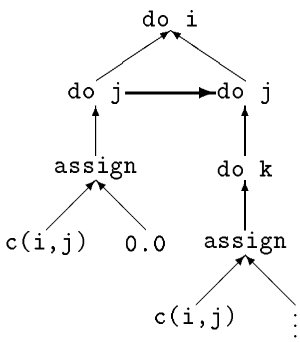


Fig. 1. The pattern hierarchy graph of Matrix Multiplication. It covers all possible ways how matrix multiplication may be coded without using an auxiliary variable.

```
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
```



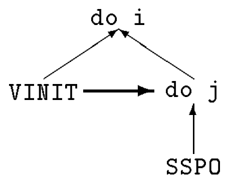
The pattern matcher traverses the syntax tree (see above) by a leftmost depth–first–search. First the assignment node $\mathsf{ASSIGN}^{(0)}$ corresponding to S1 will be replaced by the pattern instance $\mathsf{SINIT}^{(0)}(c,\ i, j,\ 0.0)$ since this is the leftmost leaf of the syntax tree (determined by the horizontal 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 $\mathsf{SINIT}^{(0)}$ node, resulting in a new pattern instance $\mathsf{VINIT}^{(1)}(c(i,*), j, m,$ $0.0)$ (vector initialization).

Then the algorithm descends further towards S2, replaces the assignment statement leaf corresponding to S2 by a pattern instance $\mathsf{AADDMUL}^{(0)}(c(i,j),$ $a(i,k), b(k,j))$ (accumulatively adding products) and, following the suitable edge in the PHG, merges this with the `do k` loop giving the instance $\mathsf{SSPO}^{(1)}(c(i,j),$

$a(i,*)$, $b(*,j)$, $k, r)$ (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 $\mathsf{SSPO}^{(1)}$ instance resulting in an instance $\mathsf{VMO}^{(2)}(c(i,*), a(i,*), b, k, r, j, m)$ (vector–matrix multiply with offset). Now the $\mathsf{VINIT}^{(1)}$ instance is a mergeable horizontal predecessor of the $\mathsf{VMO}^{(2)}$ instance. Merging them yields a new pattern instance $\mathsf{VM}^{(2)}(c(i,*),\ a(i,*), b,$ $k,\ r,\ j,\ m)$ (vector–matrix multiply), which, in turn, can be matched with the `do i` loop resulting in an instance $\mathsf{MM}^{(3)}(c, a, b, k, r, j, m, i, n)$ (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.

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. If e.g. the programmer has implemented a Gauß–Seidel relaxation in a wave–front manner, the system should choose 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 to replace it by the double number of Jacobi– or Gauß–Seidel–Jacobi hybrid iterations[8].

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

```
X[1] = A[1]
```

[8] 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.

```
      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 FOLR pattern (First Order Linear Recurrence), this piece of code will be replaced by recursive doubling techniques[9] described in [KS73].

For a lot of patterns, the programming environment supplies highly optimized parallel algorithms, e.g. for VSUM (global summation of vector elements). In these cases the parallel algorithm simply consists of a runtime library call.

For each parallel implementation of a pattern (located in a large algorithm 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 7 explains how it is determined.

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 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 ([LC90]), Knobe et al. ([KLS90, KN90]) or Wholey ([Who91]). 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. [LC90]), thus automatic partitioning may take exponential time in the worst case. That is why we intend to limit the number of distribution alternatives severely[10]. 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 [Hay93].

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,

[9] 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.

[10] In fact we will only allow the following five possibilities (for two-dimensional arrays): row-/column-/block-contiguous and row- or column–cyclic (cf. e.g. [GB90]). For all the applications considered so far these are enough to supply good distributions.

C should be aligned with either A or B. PARAMAT provides a default recommendation for alignment and one for array distribution for each pattern[11]. 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. 3.

The points between the pattern instances in the final matched program representation are natural places for possible redistribution of arrays. In [KK91] an interesting method for (static) redistribution has been presented; we will include it into the PARAMAT system with small modifications. The details will be given in a later paper.

pattern	algorithm	alignment recomm.	distribution recomm.
$A = \mathsf{MCOPY}^{(2)}(B)$	matrix copy	$A \equiv B$	arbitrarily
$V = \mathsf{VCOPY}^{(1)}(W)$	vector copy	$V \equiv W$	arbitrarily
$A = \mathsf{MJACOBI}^{(2)}(B)$	one Jacobi step	$A \equiv B$	quadratic blocks
$C = \mathsf{MM}^{(3)}(A, B)$	matrix multiply	$A \equiv C \ \vee \ B \equiv C$	A by row, B by col.
$s = \mathsf{VSUM}^{(1)}(V)$	vector sum	arbitrarily	arbitrarily
$s = \mathsf{SSP}^{(1)}(V, W)$	dot product	$V \equiv W$	arbitrarily

Table 3. Array alignment and distribution recommendations for some patterns

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. [FBZ92]).

In reality, however, it seems that these estimations very rarely meet the actual run time since a parallel computer such as the INTEL iPSC860 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 [BFKK91]. There performance estimation is based on benchmarking of simple communication patterns and propagating this information 'up' through the program's syntax tree.

Our parallelizer will be able to inspect precomputed tables of *really measured* run times for all patterns (especially, the high–level patterns!), as indicated

[11] 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. [KK91]).

above. We expect the accuracy of performance prediction being substantially improved by this method.

8 Implementation and Future Extensions

The overall PARAMAT system is outlined in Fig. 2. Note that the time–consuming work concerning parallel algorithm development (either optimized SPMD routines or just operating system calls like GLOBALSUM) 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 the special application area.

Up to now, the pattern recognizer and the PARAMAT Basic Pattern Library have been implemented. The results for the Livermore Loops and for the CG solver (cf. Tab. 2, [Lou92]) have been confirmed by the implementation. Currently, the functionality of our pattern recognition tool already exceeds the features presented in this paper (e.g., the pattern recognizer is able to perform loop re-rolling, loop unblocking and similar operations). More details can be found in [Keß94] and [Keß93].

9 Conclusions

We have outlined the PARAMAT system which performs automatic parallelization of a restricted but quite important class of numerical programs. Local parallelization is done by pattern matching and algorithm replacement. The pattern matching algorithm with the hierarchical pattern base described above is robust enough to perform its job well. Pattern matching simplifies the program graph and thus alleviates global optimizations such as determining optimal array alignment and partitioning. Pattern matching enables 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, pattern matching may be of great use at reengineering sequential dusty–deck codes for porting them to other languages or target machines.

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 iPSC860. The Pattern recognition tool and the Basic Pattern Library are already implemented.

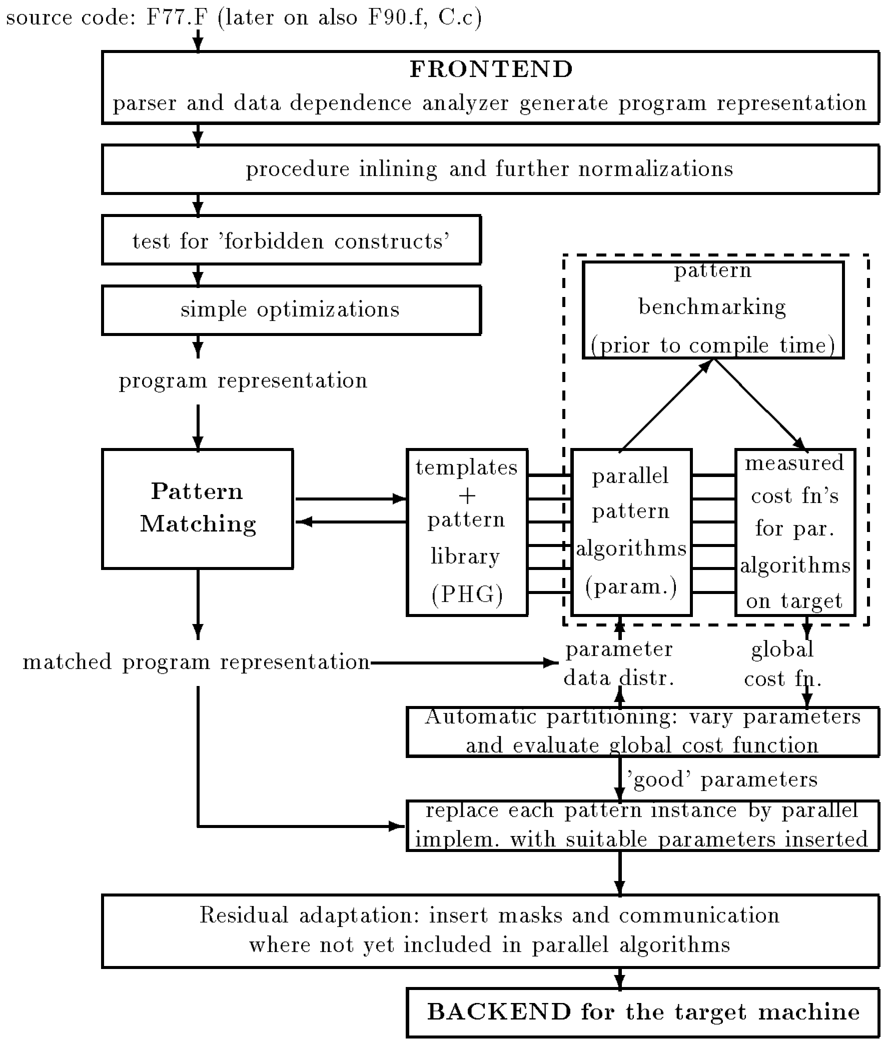


Fig. 2. Block diagram of the overall PARAMAT system. Only the components enclosed in the dashed rectangle (and the back end, of course) are dependent on the target machine. Parallel algorithms for each pattern and their run–time approximations, parameterized by the data partitionings of the arrays involved in the resp. pattern instance, have to be precomputed once for each hardware constellation (pattern benchmarking) *before* compile time.

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.

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

[CHZ91a] 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.

[CHZ91b] 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.

[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.

[FBZ92] 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.

[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.

[Hay93] Roman Hayer. Automatische Parallelisierung, Teil 2: Automatische Datenaufteilung für Parallelrechner mit verteiltem Speicher. Master thesis, Universität Saarbrücken, 1993.

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

[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.

[KK91] 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.

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

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

[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.

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

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

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

[SCMB90] 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.

[Who91] Skef Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

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

This article was processed using the LaTeX macro package with LLNCS style