Pattern–Driven Automatic Parallelization, Data Distribution and Performance Prediction

Christoph W. Keßler
Fachbereich 14 Informatik
Universität des Saarlandes
Postfach 15 11 50
D-66041 Saarbrücken, Germany
e-mail: kessler@cs.uni-sb.de

Despite the progresses made in parallelizing compilers technology during the last decade, for instance, at dependence analysis or at automatic generation of message passing code, we remain faced with three hard problems that are particularly characteristic for distributed memory parallelization:

- The efficiency of the target program heavily depends on choosing a suitable data distribution (and sometimes, also redistribution) — these problems are NP–complete [12, 11].
- a SPMD program generated semi–automatically from a sequential source program by adapting it to given array distributions must in general be transformed to be efficient (e.g., by exploiting locality) — 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. Moreover, there is no possibility to exploit the large potential of explicitly *parallel algorithms* that have been developed over the last decades for various problems on various target architectures.
- Run time prediction for nontrivial codes on real machines is a very problematic matter for reasons of network contention, message protocols, buffering, undocumented hardware features, and others.

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. At typical application codes that are reasonable candidates to be ported to distributed memory systems, we have observed that there is only a rather limited number of typical semantic operations, called *patterns*, that often occur in these programs, in particular in the time–consuming inner loops, e.g. vector and matrix operations, simple recurrences, relaxation operations, simple reduction operations, convolutions, FFT and so on [9].

Patterns are semantics–based, mostly data–parallel operations. A pattern is regarded to be a primitive with respect to mathematical properties, occurring data structures, memory access structure, array alignment preferences and run time behaviour. We have collected around 150 patterns and also typical modifications (transformations) of them, called *templates* [9].

From this observation, we construct an automatic parallelizer called PARAMAT ("PARallelize Automatically by pattern MATching") with the following key ideas:

- First, there must be a pattern recognition tool that works fast and reliably. We have proposed such a tool in [9] and we have built it [8]. Recognized patterns are replaced by a pattern *instance*, looking similar to a call to an externally defined function.
- 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 including high–level program transformations up to local algorithm replacement.

Pattern Recognition

The Pattern Recognition Tool is described in detail in [9, 8]. We just summarize here its most important features.

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.

The pattern library is hierarchically organized, representing the natural semantic hierarchy formed by the patterns. For instance, $\mathsf{SSP}^{(1)}$ (dot product) is a (direct) subpattern of $\mathsf{MV}^{(2)}$ (matrix vector multiplication), since matrix-vector-multiplication may be coded as a loop around a dot product computation. These precedence relations trigger our pattern matching algorithm by taking only those patterns as matchable candidates into account which appear in the hierarchical scope of already matched subpatterns. This makes the matching process very fast.

At the matching stage, the source program's syntax tree is traversed bottom-up and from the left to the right. A recognized node is annotated by a pattern instance. Vertical pattern matching along the edges of the abstract syntax tree works similar to standard algorithms based on tree automata. To cope with array data flow, we compute so–called *cross edges* denoting data flow relations between array operands of different pattern instances within the same program block [8]. Pattern matching along cross edges enables merging delocalized code portions and supports important normalization transformations like loop rerolling [8].

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 around 100 nontrivial patterns with around 160 nontrivial template realizations, including the normalizing transformations loop rerolling, loop distribution with scalar expansion, and IF distribution. More patterns can easily be added. For the 16 currently recognizable Livermore Loop Kernels, the prototype needs recognition times between 0.1 and 2.5 seconds (in most cases less than 0.3 seconds) on a low-end Sun SparcStation SLC, including lexing and parsing the source, unparsing and printing the result, and some diagnosis output.

Example: For the following NAS Kernel (translat. to C)

```
for (k = 1; k <= n; k++)
 for (i = 1; i <= l; i++) {
  c[i][k] = 0.0;
  for (j = 1; j <= m; j += 4)
   c[i][k] = c[i][k] + a[i][j]  *b[j][k]
                     + a[i][j+1]*b[j+1][k]
                     + a[i][j+2]*b[j+2][k]
                     + a[i][j+3]*b[j+3][k]; }
```

the pattern recognition tool rerolls the j loop and produces finally

```
MM (i, k, j, c[1:l][1:n],
      a[1:l][1:m], b[1:m][1:n], 0.000000);
```

Code Generation by Algorithm Replacement

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

Instances of recognized patterns can now be replaced by their best known parallel implementation. Such an instance contains the name of the recognized pattern and a list of the slot entries, i.e. the actual parameters (variables, constants, simple terms) that have been bound to the formal parameters (slots) during the recognition process.

Generation of standard parallel implementations For given array distributions, a *standard parallel implementation* is generated according to the well–known techniques (cf. e.g. [14]) by insertion of masks to enforce the owner–computes–rule, and by insertion of interprocessor communication to ensure that also non-local operands are available if a statement is executed.

The generated code can often be optimized (e.g., [6]): The masks could be integrated into the bounds of a surrounding loop, thus avoiding much of the overhead due to the condition evaluation. Interprocessor communication is moved to the topmost loop level (loop distribution) that is still possible without violating data dependencies. Communication is vectorized as far as possible.

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

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

Selection of parallel implementations There exist several possibilities to generate code for a pattern instance beyond the standard parallelization. The PARAMAT user may a priori control the selection process by individual code generation directives for each pattern m to select from:

(1) a *sequential implementation* (computes m on one node processor or on the host, including the necessary communication), if the debugging bit $\textsc{SeqDebug}[m]$ has been set. The parameters controlling the data distribution and the problem sizes are ignored.
(2) a *replicated sequential implementation* (sequential computation on all node processors, corresponding to the given array distributions) if the sequentialization bit $\textsc{ReplSeq}[m]$ has been set. The parameters controlling the data distribution are ignored.
(3) a standard parallel implementation for the topmost loop l occurring in the recognized subtree, if the bit $\textsc{NoReplace}[m]$ has been set. The implementation chosen for the body r of l depends on the directives for the pattern recognized for r. We omit the details of handling blocked loops for lack of space.
The effect of $\textsc{NoReplace}[p]$ is thus the same as if the loop header l would not have been recognized.
(4) a *parallel algorithm* that is not a standard implementation, if such an algorithm exists. This parallel implementation is also parameterized in data distribution and problem sizes.

These implementations are machine dependent and are parameterized in problem size and data distributions of the operand arrays occurring in the instance. They are written as C routines and macros with inline assembler for optimal usage of local processor features[1].

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

For an instance I of a pattern p, let $\textsc{NoParallel}[m](I)$ denote a boolean predicate that evaluates to TRUE iff it is, given the problem sizes and data distributions, not advisable to generate parallel code for I. $\textsc{NoParallel}[m](I)$ clearly depends on $\textsc{NoReplace}[m]$.

[1] E.g. the arithmetic pipeline of the Intel iPSC/860 node processor i860 can only be used if the program is written in machine language — the C compiler does not vectorize. Impressive performance improvements can be reached [5] by exploiting hardware features like arithmetic pipelines, dual operation mode or dual instruction mode that are just ignored by the standard compilers.

If NoParallel$[m](I)$ evaluates to True, the effect is the same as by setting ReplSeq$[m]$ to True. Thus, if the problem size of I is known at compile time and if it is small, PARAMAT will decide to prohibit parallelization if sequential execution will be faster, thus avoiding speed-down of the target program. If the problem size is not known at compile time, a suitable run time test is inserted into the generated code. — Similar run time tests can be inserted if PARAMAT is not really sure about the value of certain important program values.

Simple linear recurrences are a classical example for algorithm replacement. Usually it is implemented as a sequential loop with the form

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

which is serialized due to a loop carried data dependence as long as a standard parallelization is used. For recognized Linear Recurrences (e.g. `FOLR`, first order linear recurrence), we can apply a suitable number of recursive doubling steps [10] to gain some parallelism while taking care of growing communication overhead.

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

Local algorithm replacement is the most complex and strongest program transformation at all. It is enabled only by the availability of pattern instances. It includes all other machine specific optimizing transformations. The suitable communication routines (cf. [13]) are a basic component of the parallel algorithms. Further optimization or vectorization of messages is not required. It forms a framework to include all useful parallel algorithms that are known so far for the corresponding class of target machines (topology, granularity, communication properties). All this expert's knowledge becomes now available for the average user, although he/she does not need to care about these algorithms or machine parameters at all.

Pattern–Driven 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 on, and we know these favorite alignment and distribution strategies for each pattern implementation. This information is stored in a table [9] and can be accessed by the *data distribution driver* for each instance.

In order to keep the search space comparably low, we restrict the possible array partitionings for vectors to blocked and cyclic, and for matrices to row/col–blocked, row/col–cyclic and quadratic non–cyclic blocks. These partitioning schemes are sufficient to supply locally optimal solutions for all of our pattern implementations. Furthermore, we admit total replication of a whole matrix or vector across all processors.

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 [12], thus automatic partitioning may take exponential time in the worst case. [3] proposes a branch-and-bound algorithm for automatic partitioning. To help with the combinatorial complexity, we make use of our knowledge on favorite local partitionings as starting configurations when performing a global search for the optimal data distribution.

[3] also covers *static array redistribution* which is a NP–complete problem itself [11]. The main problem in static redistribution is that a globally optimal distribution scheme involving redistribution may even be made up of suboptimal data distributions for *all* loop nests of the program. However, [2] shows that for application programs of moderate size an *optimal* data distribution scheme can be found within a few CPU seconds using a fast 0-1 integer programming tool. This method matches our approach very well, since the pattern instances supply the required phase representation, and the run time tables deliver suitably accurate cost estimates.

Pattern–Driven Run Time Prediction

Many performance prediction approaches (e.g. [7, 4, 3]) work *analytically* by estimating the program's run time bottom-up through the abstract syntax tree, starting at the leaves of the expression trees, with an idealized model of the target machine in mind. Specific hardware features, like e.g. caches or network traffic, yield actual run times that significantly differ from the prediction. For this reason, we follow a *synthetic* performance prediction approach that has been proposed in [1].

For each pattern implementation, PARAMAT provides a table of previously measured run times with varying problem sizes and varying array distribution schemes on the target machine. The table entries for each pattern are indexed in different data distribution configurations, in the problem sizes (logarithmic scale), and in the NoReplace flag. They also depend on the NoParallel predicate. The restriction of data distribution alternatives given above keeps the table sizes moderate.

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

This synthetic run time prediction has another important advantage over the analytical approaches: it is faster, because a table lookup suffices where otherwise complex intermediate representations have to be traversed and analyzed. For instance, the Addap[3] system's automatic data distribution engine suffers mainly from slow analytical performance estimation.

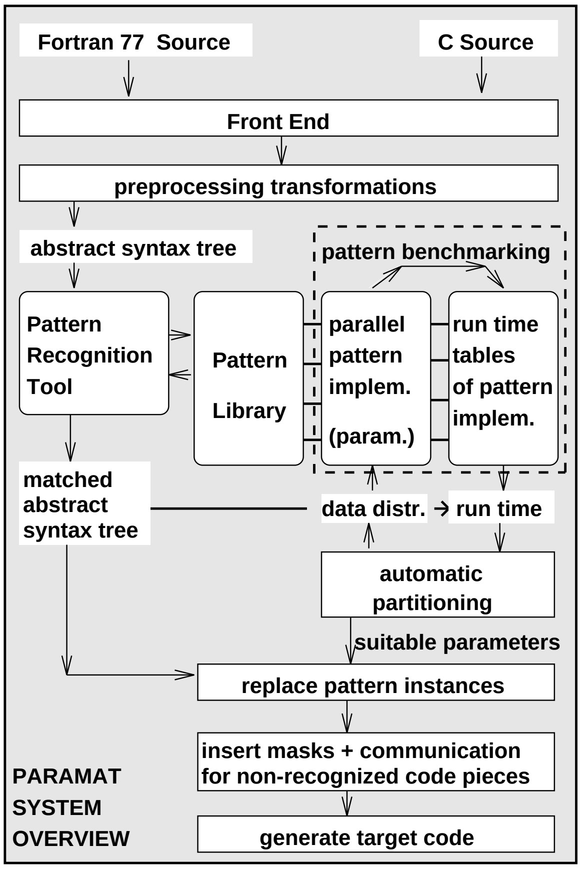

Figure 1: PARAMAT system design

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

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

For non-standard parallel algorithms the analytical methods (like [7, 4, 3]) often fail because they rely on standard parallelizations within a specific compilation environment. Synthetic performance prediction works also for all non-standard parallelizations.

System Overview

Fig. 1 summarizes the PARAMAT system design. — Note that the time–consuming work concerning benchmarking the run times of all pattern implementations for all possible array distributions and problem sizes 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.

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

Papers on PARAMAT can be obtained by anonymous ftp from `lunix.cs.uni-sb.de` in directory `/pub/PARAMAT`.

References

[1] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *Proc. ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, volume 3, pages 213–223. New York: ACM Press, April 1991.

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

[3] A. Dierstein, R. Hayer, and T. 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. Wiesbaden: Vieweg, 1994.

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

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

[6] H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, Universität Bonn, 1989.

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

[8] C. W. Keßler. Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. In K. Decker and R. Rehmann, editors, *IFIP WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verita, Ascona, Switzerland.*, pages 57–68. Basel: Birkhäuser, April 1994.

[9] C. W. Keßler and W. J. Paul. Automatic Parallelization by Pattern Matching. In *Proc. 2nd Int. Conf. of the Austrian Center for Parallel Computation, Springer LNCS 734*, pages 166–181, Oct. 1993.

[10] P. M. Kogge and H. 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, Aug. 1973.

[11] U. 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.

[12] J. Li and M. Chen. Index Domain Alignment: Minimizing Cost of Cross–referencing between Distributed Arrays. In J. JaJa, editor, *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, pages 424–433. Los Alamitos: IEEE Computer Society Press, 1990.

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

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