The Loop Parallelizer LooPo

Martin Griebl and Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany
email: {griebl,lengauer}@fmi.uni-passau.de
WWW: http://www.uni-passau.de/~lengauer

Abstract. We report on a prototype for testing different methods of space-time mapping loop nests. LooPo admits perfect or imperfect loop nests in a number of imperative languages, takes data dependences from the user or derives them itself from the source code, provides a choice of strategies for scheduling and allocating the loop nest's iterations, and produces synchronous or asynchronous parallel target code for shared-memory or distributed-memory machines.

1 Why LooPo?

LooPo is not meant to be yet another parallelizing compiler. It is a prototype system whose purpose is to assist us in the research on and evaluation of space-time mapping methods for loop parallelization. To that end, it implements the complete path from executable source code to executable target code, with switches for choosing alternative methods. At present, we provide several inequality solving methods, several schedulers and several methods of code generation, one dependence analyzer (we are working on a second) and one allocator.

LooPo is in the public domain and uses only freely available software to ensure easy distribution. It runs on Sun workstations under SunOS 4.1.x and Solaris 2.x, and on PCs under Linux.

2 Theoretical Background

The space-time mapping methods we study are based on the polytope model of nested loops, and on some extensions of it.

The polytope model supports the static parallelization of nested for loops [18]. The model represents the atomic iteration steps of d perfectly nested for loops as the points of a polytope in $\mathbb{Z}^d$; each loop defines the extent of the polytope in one dimension. The faces of the polytope correspond to the loop bounds; they are all known at compile time. This enables the compile-time discovery of maximal parallelism—relative to the choices available in the method, which are limited by the data dependences of the source program and by the requirement that the space-time mapping defining the parallelism be affine.

```
for i := 0 to n do                  for t := 0 to 2n + 2 do
  for j := 0 to i + 2 do              forall p := max(0, t−n) to min(t, ⌊t/2⌋ + 1) do
    A(i,j) := A(i − 1,j)                A(t−p, p) := A(t−p−1, p)
              +A(i,j − 1)                            +A(t−p, p−1)
```

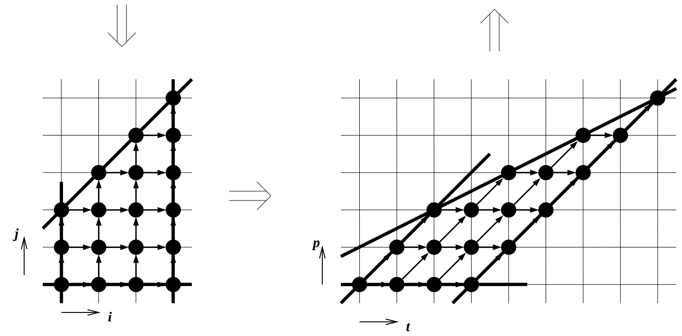

Fig. 1. Loop parallelization in the polytope model.

2.1 The basic polytope method

Parallelization in the polytope model proceeds as follows (Figure 1).

First, one transforms d perfectly nested source loops into a d-dimensional polytope, where each loop defines the extent of the polytope in one dimension. We call this polytope the *index space*. Each point of the index space represents one iteration step of the loop nest. The coordinates of the point are given by the values of the loop indices at that step; the vector of these coordinates is called the *index vector*.

Then one applies an affine coordinate transformation, the *space-time mapping*, to the polytope and obtains another polytope in which some dimensions enumerate space and the others enumerate time. We call the transformed polytope the *target space*. The standard techniques for space-time mapping in the polytope model require perfectly nested loops (all statements belong to the innermost loop body) and uniform dependences [14]. In this case, all points of the index space have identical dependence vectors which are independent of the problem size. LooPo accepts also imperfect loop nests and affine, not only uniform dependences.

Finally, one translates this polytope back to a nest of target loops, where each space dimension becomes a parallel loop and each time dimension becomes a sequential loop.

2.2 Extension to affine-by-statement scheduling

The basic model represents only perfectly nested loops. This severe restriction can be relaxed by applying the basic method to every single statement separately rather than to the body as a whole. Thus, every statement has its own index space, index vector, space-time mapping and target space [5, 9, 15]. An *operation* in the program is identified by a statement together with its index vector. Of course, the feature of statementwise space-time mapping complicates the generation of target code significantly (cf. Section 3.7).

2.3 Extension to loop nests containing while loops

One of the main future goals of LooPo is to be able to parallelize loop nests containing `while` loops. This entails a rather serious departure from the polytope model [19]. The theory and method have been worked out [12, 13], but the implementation in LooPo is still under way.

3 The Structure of LooPo

LooPo traverses a sequence of steps which transform the source program to an executable parallel target program. There are modules for scanning and parsing, (in)equality solving, dependence analysis, scheduling, allocation and target code generation. A front end provides the user with a graphical interface by which he/she can control LooPo. There is also a graphical tool for displaying index spaces and iteration dependence graphs of loop nests.

Subsequently, we give a very brief overview of the system. See our Web pages on LooPo for more details.

3.1 The Front End

The front end enables the user to invoke modules in sequence by mouse clicks on buttons (Fig. 2). Many modules provide a choice of methods via an options window, e.g., the Scheduler module (Fig. 3). Each module takes as input the output of the previous module, which is stored in a file. The user can intervene at any point by specifying hand-picked solutions in these files.

3.2 The Input to LooPo

LooPo accepts (possibly imperfect) loop nests in C, Fortran or a number of abstract loop notations, and declarations of functions, procedures and symbolic constants. By stating explicit dependences, one can experiment with the space-time mapping of non-executable programs, i.e., programs with incomplete loop bodies.

3.3 The Inequality Solvers

There are several methods for parametric linear programming, which is the central mathematical problem of the polyhedron model. We considered the following methods for use in LooPo:

Fourier-Motzkin. This is the standard doubly exponential method of polytope projection [1].

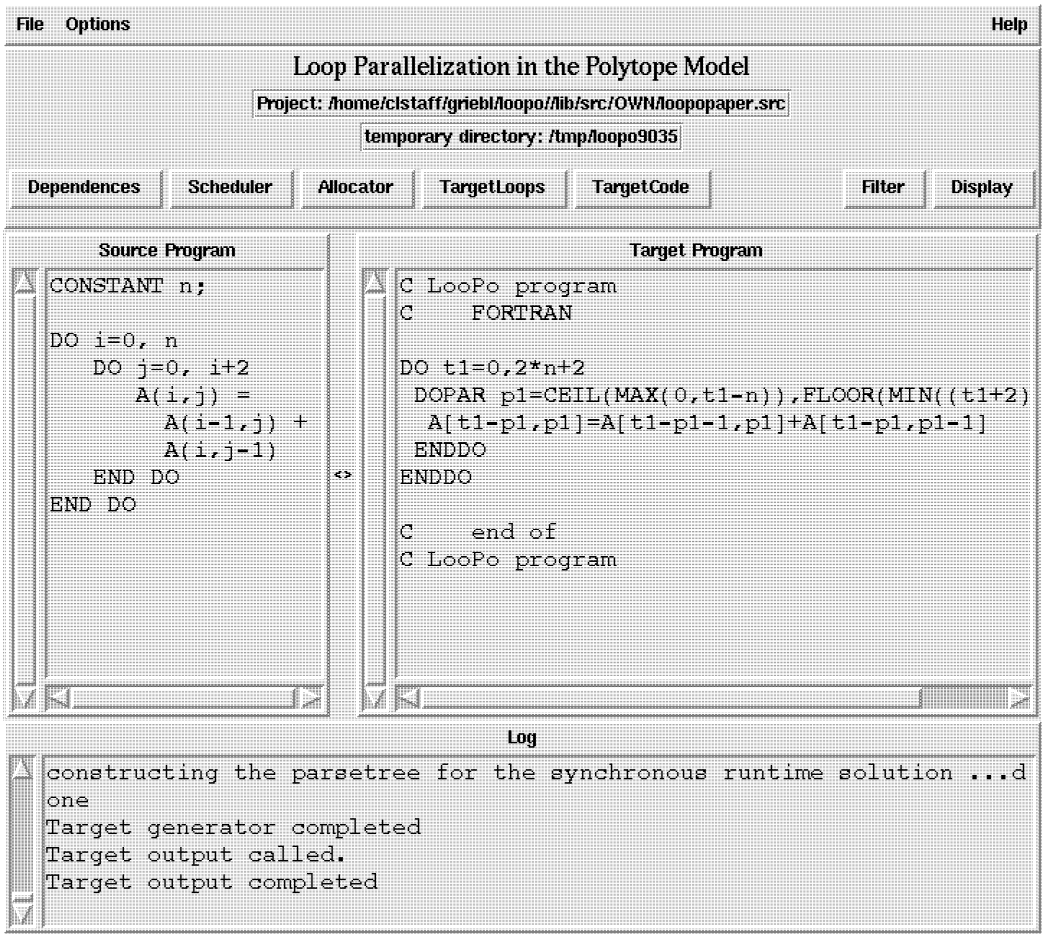

Fig. 2. LooPo's main window.

PIP. This is Feautrier's system for parametric integer programming [7]. It proceeds indirectly by transforming the original system of inequations into a dual system and solving that. In principle, it is an extension of the well-known simplex algorithm.

Weispfenning. This is another direct method which is only singly exponential [22]. It performs better than Fourier-Motzkin on problems with more than four variables.

Omega. The Omega library [21] by Pugh solves linear programs on the basis of Presburger formulas (affine constraints, the usual logical connectives, and existential and universal quantifiers), with efficient heuristics for parallelization.

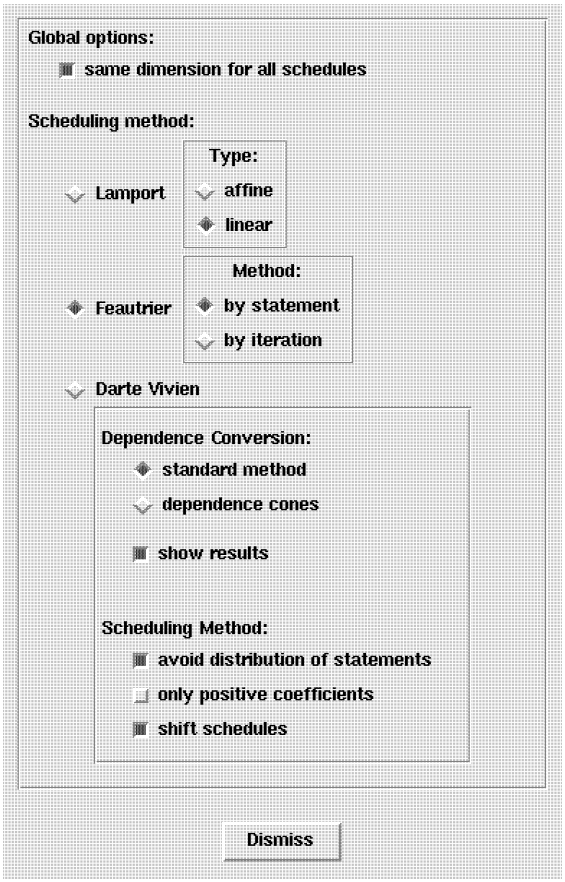

Fig. 3. LooPo's Scheduler options window.

The current implementation is largely based on PIP; the dependence module offers a choice of PIP or Fourier-Motzkin. Omega will be integrated as an alternative for PIP and Fourier-Motzkin in all modules as soon as possible.

3.4 The Dependence Analyzer

At present, LooPo features the dependence analysis method described by Banerjee [1], which makes no distinction between flow, anti and output dependences. We are working on the implementation of Feautrier's method [8], which allows potentially more parallelism, since it eliminates anti and output dependences first and then considers only the remaining flow dependences.

3.5 The schedulers

Presently, LooPo provides three different automatic schedulers:

Lamport. The hyperplane method by Lamport [2, 16] can handle perfectly nested for loops with uniform dependences. It yields a one-dimensional affine schedule for the complete loop body and, as allocation, a projection onto the source axes such that the space-time mapping formed by the combination of schedule and allocation is unimodular.

Feautrier. The Feautrier scheduler [9, 10] determines an optimal (concave) schedule for imperfectly nested for loops with affine dependences, at the cost of a longer computation time based on the necessity of dealing with parametric integer linear programming [7]. The resulting schedule for every statement can be multidimensional and piecewise affine.
For a comparison with Lamport's method, one can call the Feautrier scheduler *by iteration* (in the case of a perfectly nested input program), which enforces the same schedule for all statements in the loop body.

Darte/Vivien. Darte and Vivien proposed a fast scheduler with reasonably good results [6], which can schedule arbitrary loop programs with uniform and non-uniform dependences. It uses a less precise dependence description than the Feautrier scheduler. Therefore, the quality of its schedules is somewhere between that of Lamport's and Feautrier's schedules.

3.6 The Allocator

Presently, LooPo provides only Feautrier's allocation method [11] which determines the placement of operations on virtual processors. It is based on the "owner computes rule" and tries to "cut" dependences by mapping the depending operations to the same processor, starting with dependences in the highest dimensions (greedy heuristic). The allocator does not inspect the schedule, and may therefore generate an allocation in which some dimensions are linearly dependent. Because of LooPo's modularity, other allocators can be added easily.

3.7 The Target Generator

The target generator consists of two modules: one derives a first, raw form of the target loop nest(s), the other refines it with communications for synchronization and communication.

The target loops The loops of the transformed source program are constructed from the index spaces, the dependences, the schedule and the allocation. Note that transformations can be individual for every statement in the source program. The target loops are represented as a parse tree which does not contain any synchronization or communication statements.

The construction of the parse tree takes place in two phases. First, the statements of the source program are transformed individually. In order to achieve that, the transformation matrices are computed, the target index spaces are constructed, strides and offsets are calculated, and finally all expressions in every statement are re-indexed according to their transformed coordinates. Second, the transformed statements in their respective index spaces (*parts*) are combined to a single target program. Aside from the two options of synchronous or asynchronous code[1], three merging strategies are available [23]:

- The parts are simply combined with a *parallel operator*, i.e., there are several separate loop nests which are assumed to be executed in parallel.
- All index spaces of the parts are merged to one index space to construct a perfect loop nest. The statements inside the body are guarded by ifs to determine at run time whether or not they are to be executed.
- To avoid ifs at run time, one determines at compile time precisely in which regions which statements must be executed, and enumerates these regions successively. This is only implemented for the outermost dimension; the others are merged by the run time method. The problem is that determining the relative order of n parameters in the loop bounds requires a case distinction of order $O(n!)$ in the target program!

Synchronization and communication The parse tree representing the target loops is then translated to one of various possible output languages, e.g., parallel C, Fortran, or Parix-C[2].

Synchronization and communication is added if the user so desires. The target program (with communications) is executable on any Parix machine. We plan to add a code generator for the MasPar.

4 Related Work

There are several other systems which support space-time mapping in the polytope model:

Alpha [17] allows the space-time mapping of affine recurrence equations. The target code consists of uniform space-time recurrences, i.e., a represenation of the target polyhedron as a functional program.

Bouclettes [3] parallelizes uniform loop nests. The target code is in High-Performance Fortran (HPF).

PAF (not present in the literature as of yet) is being developed by Feautrier and implements his recent space-time mapping methods [9, 10, 11]. PAF returns, on input of an (im)perfect loop nest, a space-time mapping in the form of a (piecewise) affine expression and generates (at present, only for unimodular space-time mappings) target code in Fortran for the MasPar, Cray Y-MP and CM-2.

---

[1] In *synchronous* code the sequential loops are outermost, in *asynchronous* code innermost. Banerjee calls this horizontal and vertical parallelism [2].

[2] Parix^{TM} is an operating system for massively parallel computers with distributed memory by the Parsytec company based on the SPMD programming model [20].

There are also larger projects which are not based on the polytope model:

Crystal [4] yields target code for a number of machines (iPSC, NCUBE, Connection Machines, (i)Warp). The system imposes fewer restrictions on the input, and its use requires significantly more interaction from the user than the systems above.

SUIF [24] is the most ambitious of all projects listed here. It offers a level of intermediate code, which can be subjected to diverse program transformations; many optimization algorithms are implemented. SUIF gives highest priority not to parallelism but to the locality of data in a multi-level storage hierarchy, based in experiences with the DASH, a multiprocessor prototype.

LooPo can be used as a platform for experimenting with any step of the parallelization process in the polyhedron model; anybody interested in one special aspect of the parallelization can plug his/her own module into LooPo and obtain a customized source-to-source compilation environment. The central data structures (restricting the applicability of LooPo) are—according to the method—polyhedra and piecewise affine functions.

5 Conclusions

LooPo is a platform for comparing various parallelization techniques, based on the space-time mapping idea.

Our first tests showed that there are two main restrictions limiting the applicability of LooPo in practice. The first is the lack of conditional statements in the current version of LooPo. This will be fixed soon.

The other limitation is more deeply connected with the use of the polytope model for space-time mapping. The polytope model offers very precise analysis and scheduling techniques. However, these techniques are based on integer linear programming which is a difficult task. We have had to learn that the (in)equation solvers are the crucial factor of the parallelizer: they do not only consume most of the compilation time but even give up very frequently without computing a solution.

The implementation of the back end module for architecture-specific adaptations is still under way. Hence, we regret to be unable to present performance studies of the various implemented parallelization techniques at this time.

Acknowledgements

We thank the LooPo team: Nils Ellmenreich, Peter Faber, Max Geigl, Robert Günz, Harald Keimer, Radko Kubias, Wolfgang Meisl, Martina Schumergruber, Sabine Wetzel, Christian Wieninger and Alexander Wüst. Their spirit made this project much more than an exercise in software development. We also thank Jean-François Collard, who has been a continuous source of energy and encouragement.

This work is part of the DFG project RecuR and received travel funds from the DAAD exchange program PROCOPE.

References

1. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer, 1993.
2. U. Banerjee. *Loop Transformations for Restructuring Compilers: Loop Parallelization*. Kluwer, 1994.
3. P. Boulet, M. Dijon, E. Lequiniou, and T. Risset. Reference manual of the Bouclettes parallelizer. Technical Report 94-04, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, October 1994.
4. M. Chen, Y. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7. ACM Press, 1991.
5. A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):814–822, August 1994.
6. A. Darte and F. Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical Report 94-24, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, September 1994.
7. P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.
8. P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.
9. P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992.
10. P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, October 1992.
11. P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.
12. M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, September 1994.
13. M. Griebl and C. Lengauer. A communication scheme for the distributed execution of loop nests with while loops. *Int. J. Parallel Programming*, 23(5):471–495, 1995.
14. R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.
15. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Dept. of Computer Science, Univ. of Maryland, 1994.
16. L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, February 1974.
17. H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *J. VLSI Signal Processing*, 3:173–182, 1991.
18. C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.
19. C. Lengauer and M. Griebl. On the parallelization of loop nests containing while loops. In N. N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proc. 1st Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, pages 10–18. IEEE Computer Society Press, 1995.
20. H. Partsch. Some experiments in transforming towards parallel executability. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 71–110. Kluwer Academic Publishers, 1993.

21. W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. *ACM SIGPLAN Notices*, 27(7):140–151, July 1992. *Proc. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI).*
22. V. Weispfenning. Parametric linear and quadratic optimization by elimination. Technical Report MIP-9404, Fakultät für Mathematik und Informatik, Universität Passau, 1994. To appear in *J. Symbolic Computation.*
23. S. Wetzel. Automatic code generation in the polytope model. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1995.
24. R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *Proc. Fourth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 31–37. ACM Press, 1994.