Issues of the Automatic Generation of HPF Loop Programs

Peter Faber, Martin Griebl, and Christian Lengauer

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

Abstract. Writing correct and efficient programs for parallel computers remains a challenging task, even after some decades of research in this area. One way to generate parallel programs is to write sequential programs and let the compiler handle the details of extracting parallelism. *LooPo* is an automatic parallelizer that extracts parallelism from sequential loop nests by transformations in the polyhedron model. The generation of code from these transformed programs is an important step. We report on problems met during code generation for HPF, and existing methods that can be used to reduce some of these problems.

1 Introduction

Writing correct and efficient programs for parallel computers is still a challenging task, even after several decades of research in this area. Basically, there are two major approaches: one is to develop parallel programming paradigms and languages which try to simplify the development of parallel programs (e.g., data-parallel programming [PD96] and HPF [Hig97]), the other is to hide all parallelism from the programmer and let an automatically parallelizing compiler do the job.

Parallel programming paradigms have the advantage that they tend to come with a straightforward compilation strategy. Optimizations are mostly performed based on a textual analysis of the code. This approach can yield good results for appropriately written programs. Modern HPF compilers are also able to detect parallelism automatically based on their code analysis.

Automatic parallelization, on the other hand, often uses an abstract mathematical model to represent operations and dependences between them. Transformations are then done in that model. A crucial step is the generation of actual code from the abstract description.

Because of its generality, we use the *polyhedron model* [Fea96,Len93] for parallelization. Parallel execution is then defined by an affine *space-time mapping* that assigns (virtual) processor and time coordinates to each iteration. Our goal is then to feed the resulting loop nest with explicit parallel directives to an HPF compiler. The problem here is that transformations in the polyhedron model can, in general, lead to code that cannot be handled efficiently by the HPF compiler. In the following section, we point to some key problems that occur during this phase.

2 Problems and Solutions

The first step in the generation of loop programs from a set of affine conditions is the scanning of the index space; several methods have been proposed for this task [KPR94,GLW98,QR00]. However, the resulting program, may contain array accesses that cannot be handled efficiently by an `HPF` compiler.

This can be partly avoided by converting to *single assignment* (SA) form. This transformation [Fea91,Coh99] is often used to increase the amount of concurrency that can be exploited by the compiler (since, in this form, only true dependences have to be preserved). Converting to SA form *after* loop skewing – which is proposed by Collard in [Col94] – yields relatively simple index functions: index functions on the left-hand side of an assignment are given by the surrounding loop indices, and index functions on the right-hand side (RHS) are simplified because uniform dependences lead to simple numerical offsets and, thus, to simple shifts that can be detected and handled well by `HPF` compilers.

However, there are three points that cause new problems:

1. SA form in its simple form is extremely memory-consuming.
2. Conversion to SA form may lead to the introduction of so-called ϕ*-functions* that are used to reconstruct the flow of data.
3. Array occurrences on the RHS of a statement may still be too complex for the `HPF` compiler in the case of non-uniform dependences, which may again lead to serialized load communications.

The first point is addressed by Lefebvre and Feautrier [LF98]. Basically, they introduce modulo operators in array subscripts that cut down the size of the array introduced by SA conversion to the length of the longest dependence for a given write access. The resulting arrays are then partially renamed, using a graph coloring algorithm with an interference relation (write accesses may conflict for the same read) as edge relation. Modulo operators are very hard to analyze, but introducing them for array dimensions that correspond to loops that enumerate time steps (in which the array is not distributed) may still work, while spatial array dimensions should remain without modulo operators. In the distributed memory setting, this optimization should generally not be applied directly, since this would result in some processors owning the data read and written by others. The overall memory consumption may be smaller than that of the original array but, on the other hand, buffers and communication statements for non-local data have to be introduced. One solution is to produce a tiled program and not use the modulo operator in distributed dimensions.

ϕ-functions may be necessary in SA form due to several possible sources of a single read since, in SA form, each statement writes to a separate, newly introduced array. ϕ-functions select a specific source for a certain access; thus, their function is similar to the ?-operator of `C`. In the case of selections based on affine conditions, ϕ-functions can be implemented by copy operations executed for the corresponding part of the index, which can be scanned by standard methods. Yet, even this implementation introduces memory copies that can be avoided by generating code for the computation statement for some combinations of possible sources directly, trading code size for efficiency.

For non-affine conditions, additional data structures become necessary to manage the information of which loop iteration performs the last write to a certain original array cell. Cohen [Coh99] offers a method for handling these accesses and also addresses optimization issues for shared memory; in the distributed memory setting, however, generation of efficient management code becomes more complicated, since the information has to be propagated to all processors.

A general approach for communication generation that can also be used to vectorize messages for affine index expressions is described by Coelho [ACKI95]: send and receive sets are computed, based on the meet of the data owned by a given sending processor with the read accesses of the other processors – all given by affine constraints. The corresponding array index set is then scanned to pack the data into a buffer; an approach that has also been taken by the dHPF compiler [AMC98]. In a first attempt to evaluate this generalized message vectorization using portable techniques, we implemented `HPF_LOCAL` routines for packing and sending data needed on a remote processor and copying local data to the corresponding array produced by single-assignment conversion. However, our first performance results with this compiler-independent approach were not encouraging due to very high overhead in loop control and memory copies.

Communication generation may also be simplified by communicating a superset of the data needed. We are currently examining this option. Another point of improvement that we are currently considering is to recompute data locally instead of creating communication, if the cost of recomputing (and the communication for this recomputation) is smaller than the cost for the straightforward communication. Of course, this scheme cannot be used to implement purely pipelined computation, but may be useful in a context where overlapping of computation with communication (see below) and/or communication of a superset of data can be used to improve overall performance.

Overlapping of communication with computation is also an important optimization technique. Here, it may be useful to fill the temporaries that implement the sources of a read access directly after computing the corresponding value. Data transfers needed for these statements may then be done using non-blocking communication, and the operations, for which the computations at a given time step must wait, are given directly by an affine function. Although our preliminary tests did not yield positive results, we are still pursuing this technique.

A further issue is the size and performance of the code generated by a polyhedron scan. Quilleré and Rajopadhye [QR00] introduce a scanning method that separates the polyhedra to be scanned such that unnecessary `IF` statements inside a loop – which cause much run-time overhead – are completely removed. Although this method still yields very large code in the worst case, it allows to trade between performance and code size by adjusting the dimension in which the code separation should start, similar to the Omega code generator [KPR94]. So, the question is: which separation depth should be used for which statements? A practical heuristics may be to separate the loops surrounding computation statements on the first level, scan the loops implementing ϕ-functions separately, and replicate these loop nests at the beginning of time loops.

3 Conclusions

We have learned that straightforward output of general skewed loop nests leads to very inefficient code. This code can be optimized by converting to SA form and leveraging elaborate scanning methods. Yet, both of these methods also have drawbacks that need to be weighed against their benefits. There is still room left for optimization by tuning the variable factors of these techniques. Code size and overhead due to complicated control structures have to be considered carefully.

Acknowledgements This work is being supported by the DFG through project *LooPo/HPF*.

References

[ACKI95] C. Ancourt, F. Coelho, R. Keryell, and F. Irigoin. A linear algebra framework for static HPF code distribution. Technical Report A-278, ENSMP-CRI, November 1995.

[AMC98] V. Adve and J. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. *ACM SIGPLAN Notices*, 33(5), May 1998.

[Coh99] A. Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. PhD thesis, Laboratoire PRiSM, Université de Versailles, December 1999.

[Col94] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. of the Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G 10.3*, Caracas, Venezuela, April 1994. North Holland.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[Fea96] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.

[GLW98] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 106–111. IEEE Computer Society Press, 1998.

[Hig97] High Performance Fortran Forum. *HPF Language Specification*, 1997.

[KPR94] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report 3317, Dept. of Computer Science, Univ. of Maryland, 1994.

[Len93] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[LF98] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(2):649–671, 1998.

[PD96] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*, LNCS 1132. Springer-Verlag, 1996.

[QR00] F. Quilleré and S. Rajopadhye. Code generation for automatic paralelization in the polyhedral model. *Int. J. Parallel Programming*, 28(5), 2000. to appear.